UiO : Department of Informatics
University of Oslo

Formal Modeling and Analysis of The CANOpen Protocol in Full Maude

Wenlu Zhang

Master's Thesis Spring 2014

Formal Modeling and Analysis of The CANOpen Protocol in Full Maude

Wenlu Zhang

16th May 2014

Abstract

CANOpen is a communication protocol and device specification used in automation system. It is a commercial protocol typically related with embedded networking. Since embedded systems are widely used in automations, CANOpen does not only exist in our daily life, but also in variety of industries, such as household applications, automobiles, medical equipment, subsea facilities and so on. Some of these applications are so sophisticated that it does not allow any fatal defect, like medical equipments, and some other applications are too resource-consuming to tolerate configuration delay, such as subsea platform. Although it is not a short period since CANOpen protocol is proposed, there might still be some uncovered issues. Commercial systems tests commonly rely on the specific enviroment, which is useful to discover bugs in specific applications. But in general, it is not an efficient way to find potential issues in protocol levels. Hence, we need a method to help us solve this problem. Formal verification, providing greate contributions to software and hardware system testing, is a potential method to test a protocol itself, as well, specific applications where the protocol is applied to. Maude is formal declarative formal languages based on mathematical theory of rewrite logic. Compared to traditional programming languages, a declarative language places emphasis on modeling with naturalness, ease and preciseness. Full Maude is an extension of Maude which adds the notation of object-oriented modeling manner. In these thesis, we model some key parts of CANOpen using Full Maude, which our validation work also relies on. Since CANOpen is a complicated protocol, modeling the entire functions of CANOpen is a great task. Dividing CANOpen into two logical functions as *communication* and *controlling* parts, we focus our attention on controlling function which includes *Emergency Service (EMCY)* and *Network Managerment (NMT)*. Using Full Maude, we build the model of the controlling function based on both *equation logic* and *rewriting logic*. The resulting model is a translation from the human-language-described standard to a formal form

specification, with which we then do validation on to prove the soundness of related parts of CANOpen.

Contents

List of Figures

List of Tables

Preface

First of all, I would like to thank Ingrid Chieh Yu, who is my main supervisor for this thesis. Her advice during the entire project and her remarks on my thesis have been very helpful. Her experiences and conscientiousness guide me through many difficult phases. I am also grateful to Lucian Bentea who gives me suggestive inspirations for understanding the CANOpen protocol and building the formal model. I also want to express my thanks to Olaf Owe, Jingyue Li and Ovidiu Drugan, who share me their opinions and suggestions.

I appreciate the supports, encouragements and cares from my parents. Although they are not staying with me, time and space can never stop their loves to their son at all. I thank my friends as well who keep reminding me that difficulties will finally go by. I wish they all could have good future.

Part I

Introduction

Chapter 1

Background

An offshore platform, also known as an oil platform is a large structure with facilities to drill wells, to extract and process oil and natural gas, and to temporarily store product until it can be brought to shore for refining and marketing. The drive of exploring oil and gas offshore started over one century ago. The initial developments were simply extending its exploration and production operations with land-based rigs, wellheads and pipelines to shallow water barges. This evolution from land to sea occurred in 1897 with the first derrick placed atop a wharf on the California (USA) coast[1]. As offshore technologies advanced to conquer increasingly hostile and challenging environments, offshore drilling moved forward to two directions. First, predictably, wells were drilled at greater water depth year by year. Second, special equipments for subsea drillings entered into the water. More and more of the operations originally performed at surface are moving to the seafloor[2].

Today's subsea technology covers a wide range of equipments and activities: blowout preventer, fluid pump, water seperator, electrical power distribution system, etc. On top of the water, there are power unit supplying the energy and central computer for monitor and control of all subsea equipments. Figure 1.1 depicts a simple offshore system block diagram.

The connection between equiments and equipments to topside control facilities are based on ethernet. Typically, within each subsea equipment, there are several actuators and sensors connected by Control Area Network bus (CAN bus) system. CAN bus is a embedded bus standard designed to allow microcontrollers and devices to communicate with each other within a embedded system without a host computer.

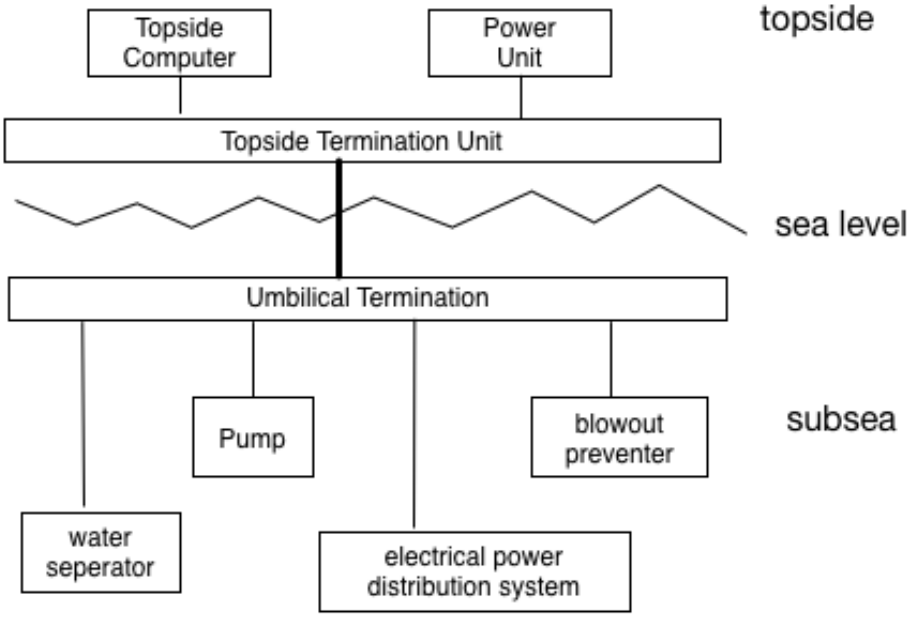

Figure 1.1: *block diagram of a simple offshore system*

Referring to Open System Interconnection (OSI) seven layer model, CAN bus provides physical layer and data link layer functions for embedded systems. For equipments to work appropriately, we also need some upper layer protocols or standards. CANOpen[6] is a higher-level communication protocol for embedded systems. It was first released in 1995 by CiA designed for automobiles. Born to be a commercial protocol, CANOpen has one big advantage that it is a framework for programmable systems as well as different devices, interfaces and applications. CANOpen extends rapidly in the industry segments, such as vehicles, medical facilities, printing machines, HVAC (heating, ventilating, air-conditioning) and so on. Some of the application areas cannot tolerate faults, such as medical system. Some of the application areas are time-sensitive, because delay may bring losses of millions of dollars each day.

The time-sensitive problem is especially obvious in the subsea industry. Subsea systems are expected to operate on the seabed without maintenance for 25 to 30 years[5]. Any error recovery or device maintenance will also affect oil and gas production. Additionally, the cost of downhole system repair is not a a small amount. Typically, the control equipment of a downhole platform is smaller than the average car[5], meaning that it is more sophisticated to fix.

Maude is a high-performance reflective language and system supporting both equational and rewriting logic specification and programming for a wide range of applications[3]. It is also a formal reasoning tool that can be used to model and analyze communications and communication networks. Communication protocols are high-level descriptions of distributed computer systems, and include broadcast and multicast protocols, database protocols, and security protocols. Such protocols are notoriously hard to design and understand due to nondeterminism and reactivity. Maude is a widely used state-of-the-art formal specification language and tool, and has been shown to be well suited to specify and analyze a wide range of sophisticated communication protocols[4]. This thesis concerns the modeling and verification of network communication in subsea equipments by Maude.

1.1 Motivation and Challenges

To control and solve time-sensitive problems in the offshore oil and gas industry, Det Norske Veritas & Germanischer Lloyd (DNV GL) is dedicated to conformance testing for subsea systems and pays much attention to the system emergency handling. DNV GL is the leading technical advisor to the global oil and gas industry, working with the sector to enable safe, reliable and enhanced performance in projects and operations. Through world-class technical assurance, advisory and risk management services, DNV GL helps customers to perform on time and to budget, even in the most demanding environments.

In addition to the general problem, DNV GL has met another issue. Equipments used for constructing a subsea system are typically provided by different manufacturers. When devices are delivered to the platform, the sooner all devices can be integrated together to work, the less deferred costs are. For the whole system to work properly, it is necessary to have a standard to conform to. CANOpen is one of the protocols which is widely applied in the offshore oil and gas industry. This thesis focuses on the application layer and communication profile of CANOpen which is describe by the standard of CiA301. So in the remaining parts, when we use CANOpen, we mean CANOpen in CiA301. CiA301 consists of a many sub-protocols of CANOpen, It depicts not only the high-level system behaviors but also each byte and bit of the protocol messages. Although all manufacturers claim their products fully comply with CANOpen, the fact is that, they cannot work compatibly.

The objective of this master project is to help DNV GL to expose more system compatible problems before equipments are delivered to the working site. The problems could be helpful in the equipment developments and testing phase. But there are some limitations related to DNV GL's problem description. They cannot provide more details on the problem, and the equipment is also a black box for them. We do not know the actual symptom, so we cannot tell what causes the problem. It may be the transition failure or the defect of CANOpen protocol. On the other hand, because of the confidential problem, DNV GL cannot provide a real case for us to study.

The only thing we know is that all manufacturers use the CANOpen standard CiA301[6] as the guide to develop their own equipments. So our analysis has to focus on the CANOpen standard itself, without considering application behaviors. Thus, our goal is to find out if there are some issues of CANOpen in protocol levels, instead of focusing on one case. These issues are commonly existing in subsea systems, and equipments abiding CANOpen standard potentially have those problems. If there are systems to be evaluated, DNV GL could then include the found issues into their verification process. Hence, potential problems could be exposed before equipments are transported to the project site.

1.2 Research Statements

The work on this thesis consist of the following main parts:

- We build a formal model of the CANOpen protocol focusing on application layer by Maude. This model is based on the CANOpen specification CiA301.
- We explore analysis technologies such as simulation and reachability analysis on the formal CANOpen model.
- We try to take advantage of the formal model in later development phase of subsea equipment, i.e. testing phase.

The motivation for our study is to help real projects in the equipments developments and testing of the subsea industry. The problem addressed in this thesis can be summarized by the following questions:

1. How can we use Maude and rewriting logic to model a complex communication protocol such as CANOpen?

2. What are the benefits of formal analysis when applied to the CANOpen protocol?
3. Considering validations, can issues discovered by our analysis be understood by the application engineer? Are the issues discovered relevant and realistic for equipment suppliers using CANOpen?
4. How can we identify the discovered issues on real systems under test?

1.3 Structure of the Thesis

To be more understandable, we will try to present our research results in a systematic way. The rest of this thesis is structured as follows:

- **Chapter 2** gives an overview of the CANOpen protocol, as well, the basic knowledge of CAN bus which are related to formal model. We introduce the CANOpen and CAN bus through a simple vehicle example.
- **Chapter 3** gives a high-level introduction of Maude, including equation logic, rewrite logic and the application of rules which are used in our formal model.
- **Chapter 4** dives into more details of the CANOpen protocol and the CAN bus system. Along with explaination of the protocol, we will show how various parts of the CANOpen protocl and the CAN bus are fomalized in Maude.
- **Chapter 5** presents the performed analysis on different parts of the CANOpen protocol both separately and together. In addition, we will show some sample test cases obtained from our analysis, which could be useful in CANOpen application development and testing.
- **Chapter 6** presents the conclusion and possible future works.

The main part of the thesis will not fully cover all aspects of the Maude model. However, Appendix A contains all technique details the complete Maude code.

Chapter 2

Overview of CANOpen and CAN Bus

In this chapter, using a vehicle example, we will give a more detailed introduction of CANOpen and CAN bus. We will explain how parts of a vehicle work using CAN bus and CANOpen protocol, then we have a more clear recognition of how CANOpen systems behave through the concrete example. Only parts of the protocol which are related to construction of our formal model will be introduced. For the full introduction of CAN bus and CANOpen, one can see [7] and [8]. CANOpen, as a communication protocol, complies with OSI 7-layers reference model of Figure 2.1. This model defines 7 layers from physical media layer up to application layer. Unlike most on-chip communication interfaces which only implement the first layer functionality, CAN bus also provides partial layer 2 functionality. As illustrated in Figure 2.1, CANOpen provides application layer functionalities and bypasses the other layers that are located between application layer and data link layer. Furthermore, it also defines the communication profiles of CAN bus applications. Theoretically, CANOpen can work properly on top of any type of network model which provides functionalities of physical and data link layer, such as Ethernet. But in practice, CANOpen is mostly applied with CAN bus.

The OSI 7-layer reference model seperates specific functionalities into different layers. To make sure exchangeability of adjacent layers, there are defined interfaces that should be obeyed by implementations. Interface specifications, bring not only interoperations of implementations in different layers, but also

overheads that are unacceptable for embedded applications. This is one of the reasons why CANOpen only implements parts of functionalities in higher layers. Before introducing the vehicle example, we will give a brief introduction of CAN bus.

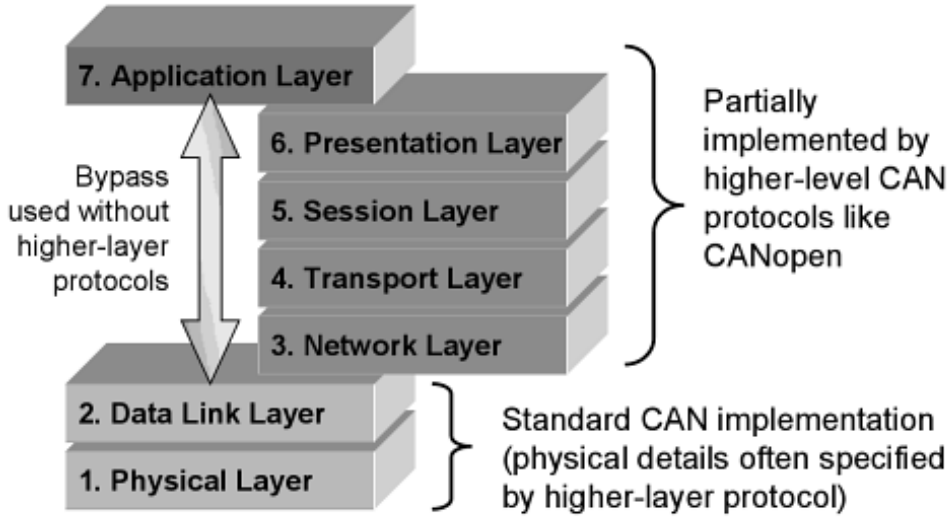

Figure 2.1: *OSI 7-layers Reference Model*

2.1 High Level of CAN Bus System

Development of the CAN bus started originally in 1983[9]. It was first officially introduced in February of 1986, by Robert Bosch GmbH at the Society of Automotive Engineers congress. Ever since low costs CAN microcontroller appeared, CAN become a prevalent network technology. Today, almost every new passenger car manufactured in Europe is equipped with at least one CAN network. Also used in other types of vehicles, from trains to ships, as well as in industrial controls, CAN is one of the most dominating bus protocols maybe even the leading *serial bus* system worldwide.

As the name suggests, serial bus implies serial communication system which is the process of sending data one bit at a time, sequentially, over a communication channel. This is in contrast to to networks we commonly see, such as Ethernet, where several bits are sent as a whole in the network. If there are several bits to be transfered in a CAN bus system, the next one cannot be sent until the preceding one is delivered. This important feature decides that

at any time, there is at most one message in the CAN bus system, since one message consists of several bits typically.

Compared to earlier automobile communication network, CAN provides a much simpler bus system. Before CAN bus was introduced, devices in the same automatic system are connected using multiple cables. Additionally, to ensure that messages could be delivered to appropriate devices, placement of different devices must be considered as well, which is less efficient and more complex. Now, one CAN bus can replace almost all cables; devices required to send or receive messages only need to be plugged into one single bus. Moreover, each message will be received in the same period by all devices connected to the CAN bus system. That means the distance between the sender and receivers will not impact messages transmission. So the problem related to the placement of devices can be completed avoided. This new feature also provides flexibility to add and remove devices in CAN bus system.

In fact, messages in CAN bus are broadcast, and all devices connected to the bus system can communication with each other. A message transfered on the bus can be received by every node. In another word, a message is related to one sender and multiple receivers. Different receivers have different physical distances from the sender, so a bit is not delivered to receivers at the same time. The hardware provides a mechanism which guarantees that each bit is delivered in a fixed period. Since data is transfered bit by bit, before the last bit of a message is delivered to all nodes, CAN bus will not allow a new message to be transmitted.

2.2 A Simple Vehicle Example

Since CAN bus and CANOpen was first designed for automobile industry, Figure 2.2 is an example of a topological structure of CAN-based network used in an automobile system.

- *Control Unit* could be s small computer which has logical process capability. It collects data from other devices, responses requests from them and manipulates them.
- *Thermal Sensor* can test temperature in vehicle and send the data back to*Control Unit*. If the temperature is too high, *Control Unit* could send control message to Ari Condition to indicate it to start to cool. Likewise,

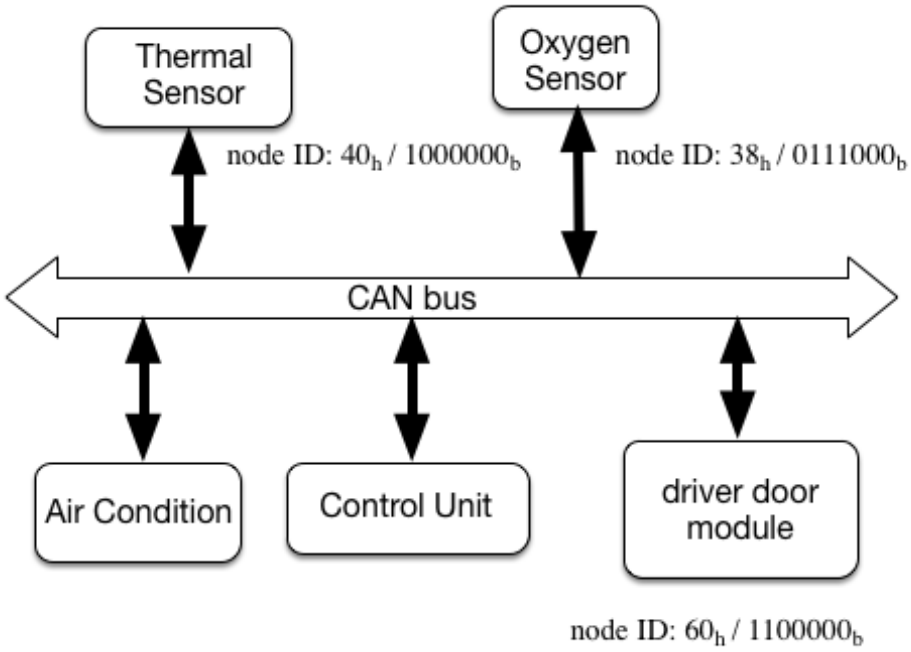

Figure 2.2: *Topological Structure of CAN-based Network in An Automobile System*

when the temperature exceeds the lower threshold, *Air Condition* then start to heat by the control of *Control Unit*.

- *Air Condition* controlled by *Control Unit*, can heat and cool the air in vehicle.
- *Driver Door Module* can detect if the door of driver position is closed. When vehicle is starting up and the driver door is not closed properly, this module will send indication message to *Control Unit*.
- *Oxygen Sensor* is a electronic device that measures the exhaust gas concentration of oxygen for internal combustion engines. Driver can also use another similar device to measure the partial pressure of oxygen in their breathing gas. *Control Unit* could request the current value of oxygen sensor, and the sensor also sends a warning when oxygen proportion exceeds some threshold.

In addition to all the devices explained above, there are several other sensors and actuators with more complex functions in real applications. Furthermore, one vehicle could have more than one CAN bus systems, and multiple CAN bus systems are gathered by CAN gateways.

2.2.1 Hardware Behaviors in The Vehicle Example

It is not rare that there are more than one devices requiring to send messages. For instances, both *Thermal Sensor* and *Oxygen Sensor* require to send temperature data and oxygen concentration data respectively. Since CAN bus only allows one message to be transferred at a time, there is a mechanism provided by hardware to determine which message could be transferred first. Actually, each message has an associated priority, which is configured in advance in configuration files. The priority is decided by both the device and the message type. In our example, temperature and oxygen concentration are regular application data, but we probably allocate a higher priority to *Oxygen Sensor* than *Thermal Sensor* in configuration files. Hence, *Oxygen Sensor* wins in this arbitration. It will take up the CAN bus until the oxygen concentration data is completed transfered. When *Oxygen Sensor* finishes sending, *Thermal Sensor* can start to send the temperature data or compete in next round arbitration. It depends if there is any new device requiring to send data.

The "failure" in the arbitration has to store its message temporarily for further transmission. In practice, this functionality is provided by *CAN controller*. The CANOpen device cannot access to the CAN bus network directly. The CAN controller acts the role of an interface between the CANOpen device and the CAN bus. Each CANOpen device has an associated CAN controller. Figure 2.3 illustrates a basic CAN controller block diagram. There are two types of buffers: one is from CANOpen device to CAN bus referred to as *sending buffer*; the other with opposite direction, from CAN bus to CANOpen device is named as *receiving buffer*. In our example, when *Thermal Sensor* fails in the arbitration, the related temperature message is stored in the *sending buffer*. As mentioned above, at any time, there is at most one message in CAN bus. So there is quite a chance that *Driver Door Module* needs to send data during other device is sending the message. The *Driver Door Module* does not have to wait, and it will continuously handle the following tasks. The out-going message is stored in *sending buffer* as well. When CAN bus is ready for the preceding messages, the CAN controller then tries to send it out.

CAN bus now supports the highest transmission rate as 1Mbps, which is not as fast as other network systems today. But considering the price, most of the microcontrollers of CANOpen devices are not so powerful that there is probably a delay of processing some of the incoming messages. It is possible

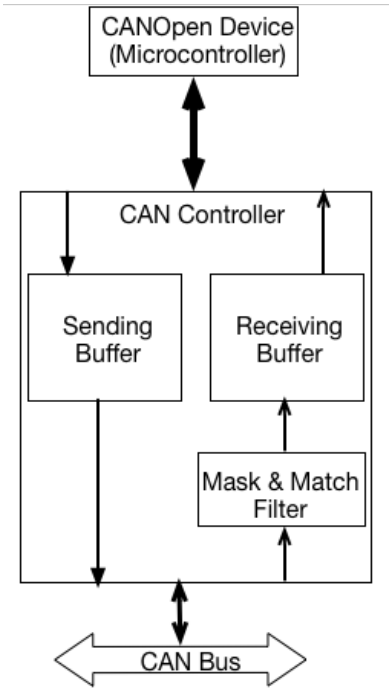

Figure 2.3: *Basic CAN Controller Block Diagram*

that all the other devices in Figure 2.2 besides *Control Unit* have delivered messages to report data, so the *Control Unit* cannot process these messages in time. The *receiving buffer* related to *Control Unit* is used to store the messages to be processed.

Since messages are broadcast in CAN bus, the regular temperature reporting messages from *Thermal Sensor* can be detected by all devices in Figure 2.2. However, only *Control Unit* in our example is interested in this message. Because of the limitation of computing capacity of microcontrollers, it is not considered as a good option that other devices also process this message then discard it. Generally, a device like *Oxygen Sensor* does not expect all messages in the system, and processing unexpected messages is an overhead cost. Thus, a filter in the CAN controller is designed to blocks unexpected message by hardware. One could configure the related control of *Oxygen Sensor* to only accept messages sent from *Control Unit*. In this way, other messages including the regular temperature reporting messages will not be delivered to *Oxygen Sensor*.

In summary, the CAN bus provides the message arbitration functionality which could avoid messages conflict. Buffers provide supports for message collision

avoidance. In addition, CAN controller also supplies the filtering capability to stop unnecessary messages consuming device computing resources. Next section, we will explain how the CANOpen protocol works in the vehicle example.

2.2.2 CANOpen Behaviors in The Vehicle Example

CANOpen describes a specification to configure and communicate real-time data as well as the mechanisms for synchronization between CANOpen devices. The functionalities CANOpen protocol supported can be logically divided over different *service objects*. Each service object offers a specific function, such as application data transmission, configuration data transmission and so on. Furthermore, a complex service may be provided by multiple sub-service objects.

All these services are classified into two groups, one is the *communication service*, the other is *control service*. As the name indicating, communication service defines the communication behaviors for CANOpen devices. Before illustrating the communication service, we will introduce another concept *Object Dictionary (OD)* which is important in CANOpen. The Object Dictionary is the heart of any CANopen device. It is essentially a grouping of objects (parameters) accessible via the network in an ordered pre-defined fashion[9]. For every node in the network there exists an OD. The OD contains all parameters describing the device and its network behavior[10]. For example, the priority configuration of different messages we talked above is also defined in OD.

According to the different transmission data, communication service includes:

- **Service Data Object (SDO)**: SDO provides the services to access Object Dictionary. Hence, in the vehicle example, the priority information could be inquired and informed by the SDO service.
- **Process Data Object (PDO)**: PDO is used to transfer real-time data; data is transferred from one (and only one) node to one or more nodes. For instances, the temperature data and oxygen data in our vehicle example are transferred by the PDO service.

Both SDO and PDO provide supports for data transmission, but for different data types. They do not fall in the scope of our formal model. [6] can provide you more technical details about how these two protocols work.

The object of our model is the control service. Compared to communication service, control service provides supports for network and system status managements for CANOpen devices. Before we introduce the control service, we will show two communication models first, which are commonly used in CANOpen protocol. It will help us to understand the CANOpen services more easily.

- **Master-Slave Model**: At any time, there is exactly one CANOpen device in the network serving as a master for specific functionality. All other CANOpen devices in the network are considered as slaves. The master issues a request to one or more slaves; addressed slave(s) acts(act) as the protocol defined behaviors. A respond message of master's request is not mandatory, depending the specific protocol.
- **Producer-Consumer Model**: The producer-consumer model is also call **Pull-Push Model**, which involves one producer and zero or more consumer(s). Producer may issue data, and any other devices which are expecting the data is seen as consumers. The consumer may or may not confirm of producer's data.

In the vehicle example, *Thermal Sensor* sending temperature messages cyclically is a procuder; the *Control Unit* is the related consumer. The *Control Unit* may or may not send acknowledgements, which is an application-specific behavior.

With the understanding of these two communication models, we can introduce the family of control service of CANOpen, which consists of the following sub-services:

- **Synchronization Object (SYNC)**: The Synchronization protocol uses a producer/consumer communication coherence. The cyclically transmitted SYNC message indicates to the consumers to start their application-specific behavior, which is coupled to the reception of the SYNC message[11].
- **Time Stamp Object (TIME)**: The Time-stamp object is used in order to adjust the global network time. The Time-stamp producer provides the current time and all network participants that consume this message adjust their local clocks according to the received timing data[12].
- **Emergency Object (EMCY)**: The Emergency Object is triggered by the occurrence of an internal error of the CANOpen device. It adopts the

producer-consumer communication model.

- **Network Management (NMT)**: Network Management Protocol defines the network status of a CANOpen device, and it also provides the control service of the network status of CANOpen devices. NMT follows the master-slave communication model. The CANOpen devcie controlling other CANOpen devices acts as the *NMT master*; CANOpen devices that are under the control of the master, are called *NMT slave*. At each time, there is only one master device in the CAN bus system.

In the vehicle example, if the *Thermal Sensor* cannot sense the temperature of the engine, or it detects the engine's temperature is over a threshold, *Thermal Sensor* could use EMCY service to report this abnormal phenomenon. After receiving the EMCY message, the *Control Unit*, the consumer of this EMCY message, could perform the pre-defined acts, such as sending the alarm to the driver.

With respect to NMT Protocol, the *Control Unit* could act as the NMT master, so other devices including the *Air Condition* are the NMT slaves. It is possible that the *Air Condition* caught some internal error, and cannot keep working properly before manual reset. So the *Control Unit* can stop the network communication of the *Air Condition*, then no more message will be received by or sent from it.

Actually, CANOpen standard consists of many sub-protocols each of which provide a specific service in CANOpen system. These services are depicted by high level explanations as well as the low level message representations in bytes and bits. In addition, the inter-connections between different services are not explicitly specified in the standard, and some of the services and system behaviors are not mandatory. These bring more difficulties to build the formal model.

The formal model addressed in this thesis mainly focuses on the EMCY Service and NMT Service. We will illustrate the details of how these two protocols work when we introduce our model. The next chapter will introduce the necessary knowledge of Maude language, which is used to build the formal model.

Chapter 3

Equation Logic, Rewrite Logic and Full Maude

This chapter will give an introduction to the formal modeling method used in this thesis. Maude was first invented by Jose Meseguer and his group at the Computer Science Laboratory at SRI International. It is a state-of-the-art formal tool in the field of algebraic specification, and it is suitable for modeling of concurrent system[14]. Maude is also a modeling formalism which supports systematic and explicit working method in analysis and verification.

Additionally, Maude is a descriptive language supporting both equational and rewriting specifications. A system can be characterized as a combination of several data, so the static system state can be specified using equational specifications. On the other hand, dynamic behaviors of a system are defined by rewrite rules which describe how system states can transit from one to another in one step. In this way, rewriting logic take equational specifications as parameters

As a state-of-the-art tool, Maude also supports to model a system in the object-oriented way. Objects can be naturally specified directly by Maude, but more widely, we use Full Maude to specify and execute object-oriented systems. Full Maude is also a specification of Maude which provides syntactic support for object-oriented concepts such as *classes*, *subclasses*, *messages*, *inheritance*, etc. In Full Maude, one can easily build a model supporting multiple inheritance and asynchronous communication through message passing in a natural way.

This section will only give a brief introduction of equational logic, rewriting logic and Maude specification, more information can be found in[15].

3.1 Equational Logic and Rewrite Logic

A rewrite logic has its underlying equational logic as parameters. According to different equational logic, there are for example, unsorted, many-sorted and order-sorted versions of rewrite logic. The unsorted signature is the superclass of many-sorted signature, and the many-sorted signature is also the superclass of the order-sorted signature.

Sort is a basic definition that should be provided before further discussion. Each sort relates to a type of data, a set of operators, some of which give notions of sorts, and some of which are used to specify operations on those sorts, and equations defining the operators.

In practical usage, it is rarely seen that sorts are not in related anyway. For example, it is nature to have a sort **Int** for the integer numbers and a sort **NzInt** for the non-zero integer, since zero can not be divisor in division. Under the many-sorted definition, these two sorts are totally disjoint. We treat sort **NzInt** and sort **Int** as two different sorts without connections. So when we need plus, subtract and some other functions, it is necessary to give the definition of both **NzInt** and **Int**, which is not a elegant way. This is why we introduce order-sorted signature.

Definition 1 (Order-sorted signature). An order-sorted signature is a triple(S, $\leq$, Σ), where S is a set of sorts, $\leq$ is a partial ordering on S, and Σ is an S* $\times$ S-sorted family $\{\Sigma_{\omega,\,s} \mid \omega \in S^*, s \in S\}$ of function symbols.

$\Sigma_{\omega,s}$ is the set of function symbols with arity ω and value sort s. We often write $f : \omega \to s \in \Sigma$ for $f \in \Sigma_{\omega,s}$. If ω is the empty word, then f is often called a constant (of sort s).

Given an order-sorted signature (S, $\leq, \Sigma$), a *variable* set X is an S-sorted family $X = \{X_s \mid s \in S\}$ of pairwise disjoint sets (that is, no variable has two different sorts: $s \neq s' \implies X_s \bigcap X_{s'} = \emptyset$), also disjoint from Σ (that is, nothing can be both a variable and a function symbol). We will often write $x : s$ for $x \in X_s$. To model a system, we use the following definition **term** to express the states of the system.

Definition 2 (Terms in order-sorted signatures). Given an order-sorted signature, (S, $\leq, \Sigma$) and a variable set $X = \{X_s \mid s \in S\}$, the S-sorted set of terms $\mathcal{T}_\Sigma(X)$ $= \{\mathcal{T}_{\Sigma,s}(X) \mid s \in S\}$ is defined inductively by the following conditions:

1. $X_s \subseteq \mathcal{T}_{\Sigma,s}(X)$ for s ∈ S; that is, a variable of sort s is also a term of sort s.
2. $\Sigma_{\omega,s} \subseteq \mathcal{T}_{\Sigma,s}(X)$ for s ∈ S; that is, a constant of sort is also a term of sort s.
3. $f(t_1, ..., t_n) \in \mathcal{T}_{\Sigma,s}(X)$ if $f \in \Sigma_{s_1...s_n,s}$ and $t_i \in \mathcal{T}_{\Sigma,s_i}(X)$ for each $1 \leq i \leq n$.
4. $\mathcal{T}_{\Sigma,s'}(X) \subseteq \mathcal{T}_{\Sigma,s}(X)$ if $s' \leq s$; that is, a term in a subsort s′ is also a term of the supersort s.
5. $\mathcal{T}_{\Sigma}(X)$ is the smallest S-sorted set satisfying the above conditions.

Definition 3 (Equations). Given an order-sorted signature $(S, \leq, \Sigma)$, a $(\Sigma-)$ equation is a triple (X, t, t′), written $(\forall X)$ t = t′, where X is an S-sorted variable set disjoint from Σ, and t and t′ are terms of the same sort; i.e., t, t′ $\in \mathcal{T}_{\pm,\int(X)}$ for some $s \in S$.

A conditional $(\Sigma-)$ equation is a $2(n+1)+1-$tuple $(X, u_1, v_1, ..., u_n, v_n, t, t')$ for $n \geq 1$, written

$$(\forall X) u_1 = v_1 \wedge .. \wedge u_n = v_n \Longrightarrow t = t',$$

such that there are sorts $s_1, ..., s_n, sinSwitht, t' \in \mathcal{T}_{\Sigma,s}(X)$ and $u_i, v_i \in \mathcal{T}_{\Sigma,s_i}(X)$ for each $i \in \{1, .., n\}$.

Definition 4 (Order-sorted equational specifications). An order-sorted equational specification is an order-sorted signature $(S, \leq, \Sigma)$ and a set E of a $(\Sigma-)$ equations and conditional $(\Sigma-)$ equations.

Now we have the definition of equational specification, which is necessary for the notion of rewrite logic.

Definition 5 (Rewriting logic specification). A rewriting logic specification (also called a rewrite theory) is a tuple $\mathcal{R} = (\Omega, E, L, R)$ where Ω is an algebraic signature (which in the order-sorted case would have the form $\Omega = (S, \leq, \Sigma)$), E is a set of equations, L is a set of labels, and R is a set of unconditional labeled rewrite rules written.

$$l : t \longrightarrow t'$$

and conditional labeled rewrite rules of the form

$$l : t \longrightarrow t' \textbf{ if } cond$$

where $l \in L$, t and t′ are terms in $\mathcal{T}_{\Sigma}(X)$, and cond is a conjunction of rewrite conditions of the form $u \longrightarrow u'$, equation conditions of the form $v = v'$ and

membership conditions $w : s$, for u, u', v, v', w terms in $\mathcal{T}_\Sigma(X)$ and s a sort in Σ.

3.2 Maude Specification

This section describes how to specify data types, equational logic and rewriting logic in Maude.

There are several benefits of using Maude for modeling. Compared to imperative languages such as Java and C++, it has the following advantages:

- Maude code is much easier to be understood, since it is closer to human language and contains no elusive terminologies, reference, for loop or memory address.
- Maude code is a program and also a specification. After translating protocol or system description into Maude specification, we directly get a executable program.
- Maude is a formalism based on algebraic, so a Maude model has mathematical meanings. We can reason a Maude model easily by following mathematical rules. If we need to prove a desired property of a system, we do not have to enumerate all possible instances.
- Maude has built-in analyzing tools, which can be used in system analysis on a formal model.

3.2.1 Equational Logic Specification in Maude

In Maude, an equational specification is a functional module which is defined with the follow syntax:

fmod *MODULENAME* is

BODY

endfm

where the *MODULENAME* is the name of the module being defined, and *BODY* is a set of declarations of sorts, function symbols, variables and euqations.

In the following part of this subsection, We will first show a simple example of Maude functional model. With respect to this model, we will illustrate how to represent sort, order-sorted relationship, functions, variables and equations in Maude.

```
fmod INTEGER is
    protecting BOOLEAN .
    sorts Zero NzNat NzNeg Nat Neg Int .
    subsorts Zero < Nat Neg < Int .
    subsort NzNat < Nat .
    subsort NzNeg < Neg .


    op 0 : → Nat [ctor] .
    op s : Nat → NzNat [ctor] .
    ops _−_ _+_ : Int Int → Int .
    op _/_ : Nat NzNat → Nat .


    var NN : NzNat .
    vars N N2 : Nat .


    eq 0 + N = N .
    eq s(N) + N2 = s(N + N2) .
endfm
```

The first line in the module body is importing another module named BOOLEAN into the current module. The imported model may be Maude predefined or user defined. Maude has several predefined modules. Some of them provide the basic data type such as integer, string, boolean, and some of provide support of composited data type like set, list. Before one starts to use these data types in a module, one nas to explicitly import them.

Key word sorts is used for defining two or more sorts in Maude. After defining data types, one can specify order-sorted relation of these data types by keywords subsort or subsorts.

The function definition is denoted by a starting key word op or ops. All the function symbols have the similar form:

$op\ f : s_1\ ...\ s_n \rightarrow s\ .$

or

$ops\ f\ g\ h : s_1\ ...\ s_n \rightarrow s\ .$

for $n \geq 0$, where f g h are introduced as function symbols, s_1, ..., s_n and s are sorts. The list s_1 ... s_n is called *arity*, and s is the value sort of these functions. In the following part of this thesis, we use *function*, *function symbol*, *operator* and *operator symbol* interchangeably. In the example above, the underscore "_" indicates the position of the member in arity. There are also functions taking no arity. Typically, this kind of function defines *constant* of a specific sort with an attribute *attr* in the end of the function. In addition, the function symbol is not always explicitly denoted, as the example below

$op\ \_\_ : List\ Nat \rightarrow List\ [ctor]\ .$

where it means a list catenating with a nature is still a list.

In Maude key words var and vars are used to declare variables. In the example above, NN, N and N2 are the variable names, and NzNat and Nat are the corresponding sorts defined before.

The last part of our example shows how to define equational logic specifications in Maude. Similar to high-level programing languages, **op** declare a function and key word **eq** defines the behavior of a function. The equations in the example have the same format as

eq t = t´ .

where both t and t´ are terms. There is another kind of equation named conditional equation with the format as

ceq t = t´ if $u_1 = v_1 \wedge ... \wedge u_n = v_n$.

where u_i and v_i $(i = 1\ ...\ n)$ are all terms. By this way, the equation eq 0 + N = N . can also be defined as ceq 0 + N = N if true . where true is a constant defined in module BOOLEAN.

3.2.2 Rewriting Logic Specification in Maude

The equation logic is used to specify the static state of a system. But there is hardly a system which is static without any state transitions. We could characterize a person as an object with attributes *name*, *marry_status* and *age*. The age of the person continuously increases as the time goes by. The name of the person and the marriage status of the person may also be changed. To describe a dynamic system or object by a model, we need a way to reflect the changes.

The problem of using equational logic specification to simulate status transitions is the symmetry property. For instance, we model a person with only *name* and *age*. It it nature that the person gets older in logic

$$\text{person}(\text{“Vein”}, 39) = \text{person}(\text{“Vein”}, 40)$$

Because of symmetry property, this equation could also be reversed mathematically.

$$\text{person}(\text{“Vein”}, 39) = \text{person}(\text{“Vein”}, 40)$$

However, this does not make any sense in reality, since no one could become younger. Hence, we could use rewriting logic specification to describe behaviors like this.

Definition 6 (Rewriting logic specification). A rewriting logic specification (also called a rewrite theory) is a tuple $\mathcal{R} = (\Omega,\ E,\ L,\ R)$ where Ω is an algebraic signature (which in the order-sorted case would have the form $\Omega = (S,\ \leq,\ \Sigma)$), E is a set of equations and membership axioms, L is a set of labels, and R is a set of unconditional labeled rewrite rules written

$$l : t \longrightarrow t'$$

and conditional labeled rewrite rules of the form

$$l : t \longrightarrow t' \ \ if\ cond$$

where $l \in L$, t and t' are terms in $\mathcal{T}_\Omega(X)$, and cond is a conjunction of rewrite conditions of the form $u \longrightarrow u'$, equational conditions of the form $v = v'$ and membership conditons $w : s$, for u, u', v, v', w terms in $\mathcal{T}_\Omega(X)$ and s a sort in Ω.

With rewriting logical specification, we can simulate the age changing of a person by form

$$\text{person}(\text{Vein}, 39) \longrightarrow \text{person}(\text{Vein}, 40)$$

Intuitively, the $\longrightarrow$ denotes the transition can only happen from left side to right side. This is a significant difference from equational logic specification.

A rewriting logic is defined in the *system* module in Maude. When we introduce equational logic in Maude, equational specifications are represented in the *functional module*. A system module is similar to a function module with the exception that keywords **fmod** and **endfm** are replaced by keywords **mod** and **endm**. Rewrite rules are defined as the form

$$\text{rl } [l] : t \Rightarrow t' \ .$$

and

$$\text{crl } [l] : t \Rightarrow t' \ if \ cond \ .$$

in the conditional case. Thus, we can write the above age changing as a rewrite rule like

$$\text{crl } [birthday] : person(S, N) \longrightarrow person(S, s(N)) \ .$$

where S is a variable with *String* sort and N is a variable with *Nat* sort. Considering the reality, a condition can be added to this rule

$$\text{crl } [birthday] : person(S, N) \longrightarrow person(S, s(N)) \ if \ N < 200 \ .$$

since no one exceeds 200 years old.

When Maude executes rewriting rules, there are 2 properties needed to be mentioned:

- *Nondeterminism* means even with the same initial state, the final result might be different after applying rewrite rules. This is because there could be more than one rewrite rules matching whose left side matches the current system states, and Maude will chose one of them randomly.

- *Nontermination* In reality, not all systems are designed to have a termination, such as bank ATM machine. When an ATM finishes the service of one, it is supposed to keep waiting for the next user instead of stopping service. Reflecting the actual behaviors of a system, the resulting model should not be terminated unless we stop it manually. Analyzing the non-terminating models may require some extra attribute constraints to limit the execution times, we will explain more in Chapter 4.

3.3 Full Maude

Maude is a very powerful formalism for distributed systems and communication protocol. It is naturally to model a distributed system or a communication protocol with the notion of objects and messages. All the objects and messages together constitute a global system. The core Maude provides one way of modeling an object as the term

$$< o : C \mid att_1 : val_1,\ ...,\ att_n : val_n > .$$

which indicates an *object* of class C whose *name* or *identifier* is o and *attributes* are att_1 to att_2 with *values* val_1 to val_n respectively. With this term, we could model a person using the following form:

$$< \text{“}Vein\text{”} : Person \mid age : 28,\ ...,\ status : single > .$$

To use this term, we need define a function symbol

$$op < \_ : C \mid att_1 : \_,\ ...,\ att_n : \_ > : Oid\ s_1\ ...\ s_n \rightarrow Object\ [ctor] .$$

where C is a class, *Object* is a sort and s_1 to s_n are the sorts of att_1 to att_n. So the class *Person* can be defined

$$\begin{aligned}
&sorts\ Object\ Oid\ .\\
&subsort\ String\ Oid\ .\\
&op < _ : Person \mid age : _,\ status : _ > :\\
&\quad Oid\ Nat\ Status \rightarrow Object\ [ctor].
\end{aligned}$$

Although it is natural to model object-oriented specifications in Maude, users would always have more expectations for Maude to support object-oriented syntax such as *class*, *message* and *class inheritance* through *subclass* directly. Maude is supposed to provide exactly this kind of support for object-oriented specification in the future. For now, we have *Full Maude* to specify and execute object-oriented systems..

Full Maude is a prototype of Maude's support for object-oriented specification and for its operations on modules and module parameterization[16]. It can reduce the programing codes of specifying rewrite rules.

We have a new module named *object-oriented module* which has the syntax as follows

$(omod\ M\ is\ ...\ endom)$

To declare a class, we have the syntax in Full Maude where the sequence of attributes does not affect the class.

$class\ C \mid att_1 : s_1,\ ...,\ att_n : s_n\ .$

Let two classes C and C′ be

$$\text{class}\ \ C\ \mid\ att_1 : s_1,\ \ ...,\ \ att_n : s_n\ \ .$$

and

$$\text{class}\ \ C'\ \mid\ att'_1 : s'_1,\ \ ...,\ \ att'_n : s'_n\ \ .$$

respectively. We could define C as a subclass of C′ using the form $\text{subclass}\ C < C'$ to indicate that each object of C class is also an object of C′ class.

For communication between objects, Full Maude also defines the syntax

$$\text{msg}\ f : s_1,\ \ ...,\ \ s_n \rightarrow \text{Msg}\ .$$

to define sort *Msg*. Both *class* and *Msg* are subclasses of *Configuration*, a predefined sort in Maude.

With the notions of *Msg*, *class* and *Configuration*, a distributed system can be defined as "soup" consisted of different kinds of classes and messages. The "soup" itself also has the sort of *Configuration*.

$$\begin{aligned}&\text{op}\ \_\_ : \text{Configuration Configuration} \rightarrow \\ &\qquad \text{Configuration [ctor assoc comm id: none]}\ .\end{aligned}$$

In the above function declaration, in addition to property *ctor*, there are also three other properties. *assoc* means associativity, *comm* is the abbreviation of commutativity and *none* is a constant of sort Configuration having the similar meaning as *null* in programing language C which denotes an 'empty' sort Configuration.

3.4 Analysis Method in Maude

Constructing a model is only the first step, the final goal is what we can derive from the model. Thus, there must be ways to analyze the model and evaluate the result. The checking procedure can be performed by Maude automatically, as long as a desired property is explicitly specified.

Maude supports a wide range of formal analysis methods, including *rewrite* for simulating system behaviors, *search* and *narrowing*[17] for reachability problem, *linear temporal model checking*[18] for temporal logic properties and so on. We will take a deep look into *rewrite* and *search* methods which are used in the thesis.

3.4.1 Execution

Maude has a basic command *red* taking a term `t` as the input `red t` to reduce the given term `t` into a *normal form*. Intuitively, the procedure of reduction is applying equation specifications on a term. A *normal form* of a term t is to reduce t 0 or more times until there is no reduction can be taken. For a specification E, if each term can be reduced into its normal form, then we can say that the specification E is *terminating*.

There is also another essential property of a specification E which needs our attentions when we build a model. For a given term t, there may be more than one equation specifications can be applied for reduction. Furthermore, if a specification E is termination, then there probably exist one or more normal forms for a term t, depending on the equation specifications and the sequence of them we choose. Hence, we define a specification E is *confluent* if and only if for any term t, it has a unique normal form.

Maude does not check the termination and confluence properties of a model, so we have to make sure our models are terminated and confluent when we construct it. Otherwise, you cannot expect Maude give a satisfied result out. This article does not cover the methods to build a confluent and terminated model, you can get more information in [14].

The Maude commands `rew` and `frew` are used to simulate the possible behaviors of a system by applying rewrite rules from an initial state: `rew [n] init` and `frew [n] init` where `init` is a term denoting the system initial state and "`[n]`", as an optional parameter, means how many rewrite steps do you intend to perform.

In the previous 'Person' example, if "Vein" is still single and 28 years old and we need to check what will "Vein" become by applying 20 steps of rewrite, we can run the command

rew [20] < "Vien" : Person | age : 20, status : Single > .

There is one thing you should keep in mind that when you define rewrite rules in Maude, the left side must be a normal form. This is because before Maude perform rewrite rules on a given term, it will reduce the term to normal form first. It is not as "intelligent" as us that can tell if a reduced term matches another term which is not in its normal form.

3.4.2 Reachability Analysis

Search uses a breadth-first strategy to explore all possible behaviors from a given initial state. We use the following syntax for an unconditional search

search *init arrow pattern* .

and the syntax for a conditional search

search *init arrow pattern* such that *cond* .

where *init* is the term of initial state, *pattern* denotes the term of desired system state and *cond* indicate the search condition. There are four possible forms of the *arrow*, and each of them has different meanings:

- ⇒1: states which can be reached in exact one step
- ⇒*: states which can be reached in zero or more steps
- ⇒+: states which can be reached in one or more steps
- ⇒!: states which can not be further rewritten

Search tries to find all the possible results matching the pattern under the specified condition. But sometimes the number of results is infinite. If so, the search command will keep running without halt. Besides, not all specification are supposed to be terminating, so Maude define two other parameters with search command.

One can assign an upper bound n on the search results by the syntax

search [n] ... .

and set an upper bound d on rewrite steps by the syntax

search [,d] ... .

since Maude uses bread-first way as the search pattern. Of course, one can set these two upper bounds at the same time

search [n,d] ... .

In the thesis, we verify the soundness of the CANOpen protocol to see if there is any unexpected state existing in CANOpen systems. Given an initial state of a system, we will mostly use the reachability analysis method to check if the system could enter into a state which is not specified in the CANOpen protocol. In addition, we use frew and frew commands to demonstrate that our model could execute properly.

Part II

Modeling and Analysis

Chapter 4

Technical Details of CANOpen System and Formal Model of the CANOpen Communication Protocol

Chapter 2 gives the high-level description of CAN bus and CANOpen protocol using a vehicle example; chapter 3 introduces the basic knowledge of Maude used to build the formal model. In this chapter, we will explain the concerned details of CANOpen protocol and the CAN bus , along with which, we will also show how to formalize these details by Maude.

Requirements and specifications describing a protocol or system are the foundations of constructing a model. The actual model-building procedure is to translate the requirements and specifications into model language. One problem of the translation is that, CANOpen is a very complex communication protocol, and there are many services composing the protocol. For most of these services, there are also some optional behaviors which are not mandatory. So the protocol has a large configuration space. In the other hand, the interactions between different services are not completely covered by the standard CiA301. In addition to high-level specification of the explicit behaviors of CANOpen devices, CAN bus standard and the CANOpen protocol also contain many details in bit-wise level. This also makes the protocol complicated.

Modeling of a complex communication system like this is a big challenge. Because of the time limitations, we cannot simulate every service in the

protocol. We have to choose the modeling parts where potential problems might exist. Another difficulty is to abstract the system in a proper level without too much details. Meanwhile, the abstract will not negatively affect the analysis result.

Our model mainly focuses on parts of the underlying-layer services of CAN bus system and parts of the control services of CANOpen protocol. The underlying-layer services include the priority arbitration of the CAN bus and the message filtering and buffering of the CAN controller; the control services in the model covers the NMT protocol and the EMCY protocol of CANOpen. The rest parts of CANOpen such as SDO, PDO, TIME Objects and SYNC Objects are not parts of our model. Choosing to model the EMCY protocol and NMT protocol instead of others is because they manage the application internal state and the network state of the CANOpen device. They impact and decide the behaviors of other services. Another reason is that DNV GL also finds some issues in control service. There is supposed to be more possible issues. In our model, we ignore all of the low-level bit-wise details and simplifies some complex behaviors. The model building stays true to the CANOpen standard[6] without any implicit assumption.

4.1 Underlying-layer Behaviors of The CANOpen Sytem

Instead of introducing the underlying-layer behaviors in a high-level fashion in chapter 2, we will in this chapter consider the details about how they work and how they are represented in Maude.

4.1.1 CAN Controller

Chapter 2 introduces two important features filtering and buffering in the CAN bus communication provided by the CAN controller. In this section, we will describe the details about these two functionalities and how we model them by Maude. Even though our objective is to formalize CANOpen, we need to include CAN bus in the formalization in order to capture the feature of message transmission in CANOpen system.

4.1.1.1 Buffer

Buffers provide supports of buffering both incoming and outgoing messages. But the mechanism about how the buffers work is not included in the CANOpen protocol or the CAN bus specification. In practice, the mechanism is adopted and implemented by the manufactures of CAN controllers. Because of the cost, a CAN Controller typically has a limited number of buffers. So the essence of a buffering mechanism is how to push new messages into and pop old messages out of a buffer. For an industry production, it usually takes one of the following mechanisms:

- **FIFO**: First-In-First-Out(FIFO) is a directive and basic way. There are 2 buffers in the controller which can store multiple messages. Outgoing messages are stored in the sending queue in the order of their generations in the microcontroller. When CAN bus is ready for the next round transmission, the first message in the queue will be sent out. Likewise, incoming messages through filter are also stored in the sequence of being delivered from the bus. CANOpen device will fetch and process the first message in the receiving buffer queue. If the buffer is full, new messages are going to be discarded. By FIFO mechanism, a high priority message will need to wait until all preceding messages are processed. This is not in accordance with the intention of high priority messages first. Even worse, under the some circumstances, if buffers are full, high priority messages could be discarded directly. Consequently, these message will never be send out or delivered.

- **High Priority First**: Since some messages are so crucial like reporting fatal errors that they should be offered high guarantees to be not lost, high priority first schema is adopted by some manufactures. As the name suggests, messages in the queue with high priorities will be processed first. Moreover, when buffers are full, new messages with higher priority than any of messages in the buffer will replace the message with lowest priority. This schema solves the problem caught in FIFO, but it also brings more control complexity.

- **Full-CAN**: The early CAN controller used the so-called "Full-CAN" implementation. Full-CAN controllers have a number of *messages objects*, each of which has only one buffer for one message. Each message object is bi-directional(can be configured to either sending or receiving), and each is associated with one filter. This allows us to configure the message

object to accept only one specific message. The Full-CAN controller is very efficient as long as the number of messages the CANOpen device expecting is small. With the increasement of the scale of the CAN-based system, more messages are expected to be received. So there is no true benefit of Full-CAN controllers. In the vehicle example of Figure 2.2, *Control Unit* may listen to all messages from other devices, so it needs a great amount of message objects. Moreover, in a back-to-back scenario, each message object owns a single buffer and matching incoming message will override the buffer's content.

- **Advanced**: An improvement of the CAN controller solution is a combination of Full-CAN controller and FIFO(or high priority first). Instead of storing only one message, each message object has the capacity of multiple messages. Although powerful, it requires the most complex control mechanism.

In our model, we adopt the FIFO mechanism for buffers. The buffer is modeled by the sort `Que`, and the sort `NeQue` indicates a non-empty buffer. To insert and remove messages from the buffer, we define two equation rules named `addTo` and `pop` respectively.

```
sorts NeQue Que .
subsorts NeQue < Que .

op _addTo_ : X$Elt Que -> Que .
eq E addTo Q = Q : E .

op pop : Que -> Que .
eq pop(nilQue) = nilQue .
eq pop(E) = nilQue .
eq pop(E : NEQ) = NEQ .
```

`nilQue` is the constant representing an empty buffer. `E`, `Q` and `NEQ` are all variables which represent an element, a buffer and a non-empty buffer respectively. `addTo` ensures the new message is inserted to the end of the buffer; `pop` removes the first message of the buffer. It complies with the FIFO mechanism.

4.1.1.2 Filter

A filter is designed to block unexpected messages. To filter message, two types of registers are related. One is *mask register*, ignoring some of bits; the other is *match register* only allowing matched messages to come through. A bit being 1 at mask register will enable comparison between the bit of the match register and received message in the corresponding positions. A bit being 0 at mask register means it does not matter what is the corresponding bit in message data. We use the example in Figure 4.1 to explain how the mask and match registers work. The combination of mask registers and match registers is effective, and it could percolate most of unexpected messages and alleviate the pressure of microncontrollers.

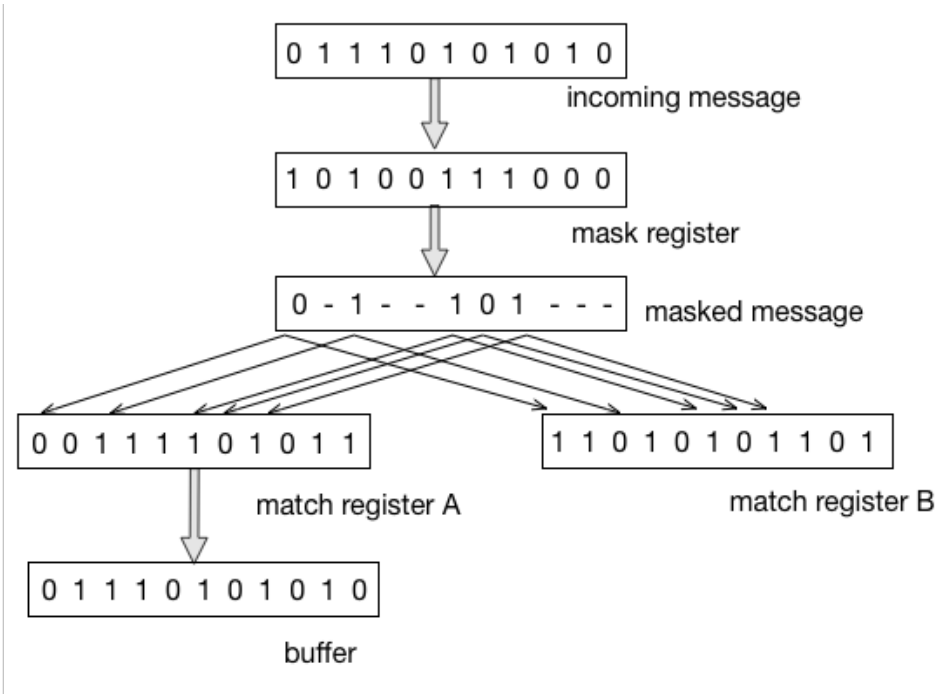

Figure 4.1: *A Simple Example of Filter Work Flow*

The filtering is an auxiliary function in CANOpen communication network, and the purpose of a filter is to percolate unnecessary messages. So we simplifies the filtering procedure. In stead of implementing the mask and match registers function, we define an equation:

```
op _percolate_ : NatSet Nat -> Bool .
eq nilSet percolate N = false .
```

```
eq (N NS) percolate N2 = if N == N2 then true
                         else (NS percolate N2)
                         fi .
```

NS is a set of natural number; N and N2 are both natural number. We configure all expected COB-IDs in the set. If the COD-ID of the incoming message is in the set, this message will be delivered to the related microcontroller. Otherwise, it will be discarded.

4.1.1.3 Model CAN Controller

With the model of filtering and buffering functionalities, we can model a CAN controller as follows:

```
class Controller |  txbuf : Que, rxbuf : Que,
                 filter : NatSet, rxbufCap : Nat,
                 txbufCap : Nat .
```

where txbuf indicates the outgoing messages buffer, rxbuf indicates the incoming messages buffer, filter is a set containing COB-IDs of all expected messages, rxbufCap indicates the size of the rxbuf and txbufCap indicates the size of the txbuf.

The behaviors of the CAN controller include frames transceiving between CAN bus and CANOpen device. We will show some of the Maude codes about the CAN controller behaviors after CAN bus model is introduced.

4.1.2 Priority Arbitration

The priority arbitration is a important functionality provided by CAN bus. It is the foundation base on which all communications in a CANOpen system can work properly. Before we take a deep look at priority arbitration, a few other technique details should be illustrated first.

4.1.2.1 CAN Frame

In a CANOpen system, each CANOpen device is configured with a unique identifier referred to as *node ID*, which is 7-bits length. Thus, in a single CAN bus, there are at most 128 nodes. An extension of CAN bus standard allows longer bits *node ID*, so that more devices can connect to a CAN bus system. But in practice, CAN-based systems using an extension are rarely seen, so we adopt the 7-bits *node ID*. Figure 4.2 indicates a general basic CAN frame. Typically, a CAN frame is composed of control data and payload. Control data has a fixed length, and payload has flexible length between 0 to 8 bytes. In Figure 4.2, we only describe parts of the control data, since others are not related to the further model and analysis.

Componet	COB-ID	RTR	Data Length	Data
Length	11 bits	1 bit	4 bits	0 - 8 bytes

Figure 4.2: *Basic CAN Frame Structure*

- **COB-ID**: *Communication Object Identifier(COB-ID)* is used to identify one message. It could also be seen as the message ID. Different COB-IDs represent different messages, and a message with smaller COB-ID has higher priority. Message priority is decided by two parts: message type and the sending device *node ID*. Messages in CAN-based system are classified, and each type of message is used to transfer a specific information. Before devices can start to communicate, every type message is assigned to a 4 bits *function code*. For example, in Figure 2.2, the *Thermal Sensor* with *node ID* 40_h is supposed to report temperature data to *Control Unit*. The vehicle system is configured to use *function code* B_h to indicate this type of message. So the COB-ID of this message is $B40_h$. That is, the first number denotes *function code*, and the last two numbers denote *node ID*.
- **RTR**: In the vehicle example, the *Thermal Sensor* can report temperatures actively driven by time. Additionally, the *Control Unit* could also request the *Thermal Sensor* to send temperature data. In CAN bus system, messages requested data are called *Remote Transmission Request(RTR)*. The

flag RTR in a CAN frame is used to denote if this message is a request or response.

- **Data Length**: *Data length* indicates the length of payload in the message. Since a CAN frame can only contain at most 8-bytes data, we only need four bits to record the data length.
- **Data**: *Data* field is the payload of a CAN frame. The payload is empty for a RTR message. The content of the payload might be application level data such as temperature and oxygen concentration, or system configuration data.

A CAN bus frame is modeled by a Maude class simply as

```
class Frame | id : Nat, other : FrameData .
```

where the attribute `id` represents the COB-ID and the attribute `other` indicates other parts of a CAN frame. `FrameData` is a predefined sort type, which is a supperclass and will be inherited by other classes. We abstract the real CAN frame into two attributes, which avoids the redundancies such as *Data Length*.

4.1.2.2 CAN Bus

Our model does not require or rely any physical features of the CAN bus. So the model of CAN bus is more like a "logic device" which records the condition of the underlying communication network. The class in our model to model CAN bus is also very simple with only two attributes,

```
class Bus | status : BusStatus, owner : Oid .
```

where attribute `owner` denotes the node which is occupying the bus to transfer data and attribute `status` denotes the current state of the bus. `Oid` is a Full-Maude predefined sort representing the object ID. `BusStatus` defines all possible states the CAN bus could stay in:

- *free*: No message is currently in the bus and no node intends to send a message.
- *started*: There are more than one nodes need to send message at the same time. So the bus is going to start a arbitration.
- *arbitrated*: The bus has "chosen" the winner in the arbitration, and the arbitration procedure is over.
- *transmitting*: The "winner" starts to transmit the message, but the last bit of the message is not delivered yet.
- *transmitted*: The message transmission finishes, and the bus is going back to free again.

4.1.2.3 CAN Bus States Transition

Figure 4.3 shows the states transition of CAN bus.

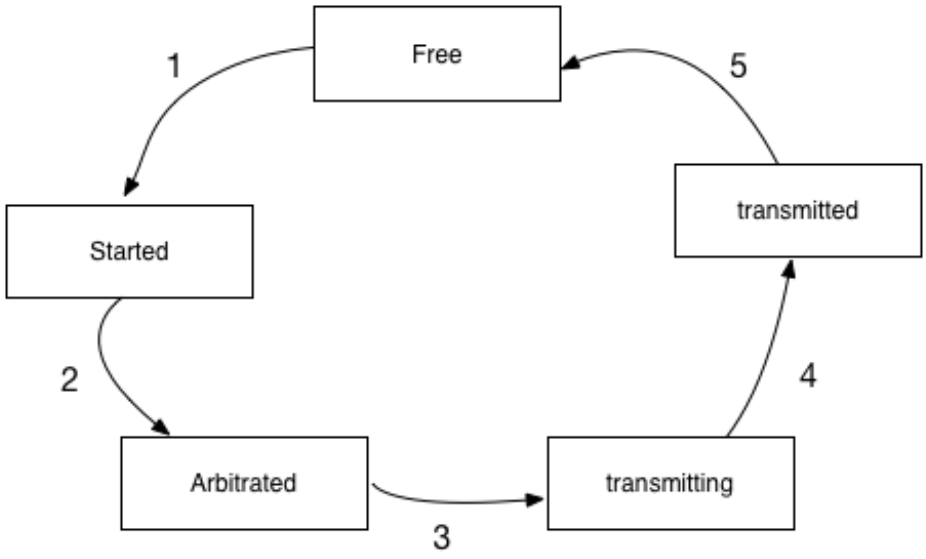

Figure 4.3: *CAN Bus States Transition*

1. When the CAN bus is *free*, any CANOpen device could use the bus to transfer data. The bus will remain in *free* state if no device needs to transfer data. If one or more devices are going to send messages, the CAN bus then enters into *started* state.

2. In the *started* state, a decision that which node can take up the CAN bus is made. If there is only node requiring to send data, the *started* state is over. The node will start to send the data and the CAN bus goes into *arbitrated* state. If there are more then one nodes having messages to transfer, then an arbitration procedure starts. Finally, when the node with the message of highest priority is selected, the bus enters into the *arbitrated* state.

3. The CAN bus will automatically enter into the *transmitting* state.

4. When the last bit of the message is delivered, the CAN bus enters into the *transmitted* state.

5. The CAN bus in *transmitted* state can automatically become *free*, where no node is occupying the bus.

We use a rewrite rule to model the transition *1* in Figure 4.3:

```
rl[startArbitration] :
GLOBAL(< BID : Bus | owner : nilOid, status : free >
        < CID : Controller | txbuf : NEBUF >
        CONF)
=>
GLOBAL( arbitrate (
        < BID : Bus |   status : started >
        < CID : Controller | >
        CONF) ) .
```

The left side of the rule denotes that if the CAN bus is in *free* state and there is at least on CAN controller with an non-empty `txbuf`, then the bus starts an arbitration procedure.

The CAN bus state transition *2* is modeled by a Maude operation `arbitrate`. We we describe the CAN bus arbitration and the operation in next section.

We model the CAN bus state transition *3* using another rewriting rule:

```
rl[transmitMsg] :
GLOBAL (
```

```
        < CID : Controller | txbuf : (< FID : Frame |
           id : COD, other : DATA > : BUF) >
        < BID : Bus | owner : CID, status : arbitrated >
        CONF )
=>
GLOBAL (
        < CID : Controller | txbuf : BUF >
        broadcast (DATA withID COD from CID) in (
                < BID : Bus | status : transmitting >
                CONF) ) .
```

In the left side, the CAN bus state `arbitrated` indicates that the arbitration is over. The attribute `owner` denotes the ID of the CAN controller which can take up the CAN bus. In the right side of this rule, the controller broadcasts its message and the CAN bus state changes into `transmitting`.

The operation `broadcast` in the above rule triggers the CAN bus state transition *4*. The operation and related equations are defined as follows:

```
op broadcast_in_ : Msg Configuration -> Configuration
   [format (n nt n nt n)] .
eq broadcast (DATA withID COD from CID) in (< CID :
   Controller | > CONF)
   =
   < CID : Controller | > (broadcast (DATA withID COD
      from CID) in CONF) .
eq broadcast (DATA withID COD from CID) in ( < BID :
   Bus | status : transmitting, owner : CID > )
   =
   < BID : Bus | status : transmitted, owner : nilOid >
      .
eq broadcast (DATA withID COD from CID) in ( < CID2 :
   Controller | > < BID : Bus | owner : CID, status :
   transmitting > CONF )
   =
```

```
(deliver (DATA withID COD from CID) to < CID2 :
   Controller | >) broadcast (DATA withID COD from
   CID) in ( CONF < BID : Bus | >) [owise] .
```

As the name suggests, this operation tries to broadcast one message to the global CANOpen system. The first equation indicates that the message is only delivered to all other CAN controllers except for the sender. The left side of the second equation indicates that all controllers have received the message. So the message broadcasting is over, and the CAN bus state enters into `transmitted`. Meanwhile, the attribute `owner` of CAN bus becomes `nilOid` which denotes no device is occupying the bus. The last equation describes that, the message is delivered to one CAN controller in the global system, and the broadcasting continues in the rest of the CANOpen system.

The last transition of CAN bus states is modeled by a simple rule:

```
rl[backToIdle] :
GLOBAL (
        < BID : Bus | owner : nilOid, status :
           transmitted > CONF)
=>
GLOBAL (< BID : Bus | status : free > CONF) .
```

which means the CAN bus will automatically switches into `free` state from `transmitted` state.

4.1.2.4 Priority Arbitration

In chapter 2, we have showed that there will be a collision when both *Thermal Sensor* and *Oxygen Sensor* having messages to sent. To solve this problem, CAN bus defines the priority which can be fully decided by COB-ID. Messages conflicts solving is one of the core features in CAN. It ensures that higher priority messages will always be delivered first, so no bandwidth is lost.

Since a lower COB-ID has a higher priority, the message with the smallest

COB-ID could win in the arbitration. All COB-IDs are compared bit by bit from left to right, the highest significant bit first. With the Figure 2.2, we still use the vehicle example to illustrate arbitration procedure. We assign node IDs 1000000_b, 1100000_b and 0111000_b to *Thermal Sensor*, *Driver Door Module* and *Oxygen Sensor* respectively. These three devices transfers corresponding application data by the same type of message with the *function code* 1110_b. So the three message COD-IDs are listed as in Figure 4.4. The first four bits are equal since they are the same type message. We begin to compare the 7 bits node ID. Obviously, 0 is smaller than 1, so *Oxygen Sensor* win the chance to transfer data.

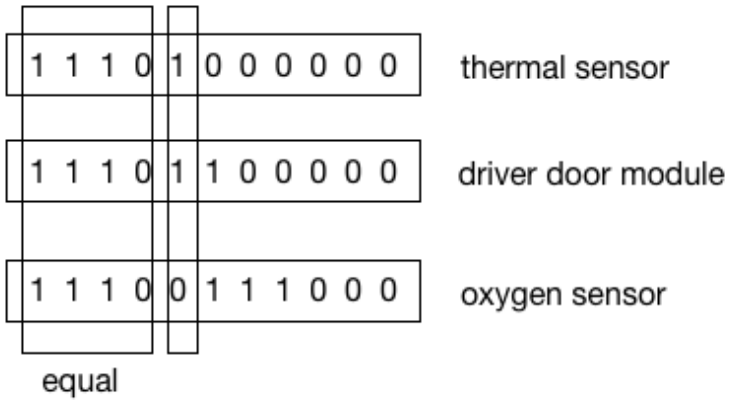

Figure 4.4: *Example of COB-IDs Comparison*

Actually, binary 0 and 1 map to different voltage in the bus. 0 corresponds to the *dominant*[19] voltage and 1 corresponds to the *recessive*[19] voltage. First, devices try to send the highest bits of their COD-IDs. The devices with *dominant* bit can continue the next round comparison. If all bits are *recessive*, no device will be eliminated. The winers in the last round bit comparison then try to send the second highest bits, and the comparison procedure is the same. The arbitration will not stop until there is only one winer left.

We model the arbitration procedure by operation `arbitrate`, which also triggers the CAN bus state transition 2 in Figure 4.3.

```
op arbitrate_ : Configuration -> Configuration [format
   (n d n)] .
```

Although this function has the name “arbitrate”, it does not always start an arbitration in the formal model. In fact, if there is only one node with messages to send, the CAN bus enters into *arbitrated* state directly and the node will be the owner of the bus.

```
eq arbitrate ( < CID : Controller | txbuf : NEBUF > <
   BID : Bus | status : started > )
=
< CID : Controller | > < BID : Bus | status :
   arbitrated , owner : CID > .
```

If there are multiple controllers with messages to send, the arbitration will take two of them and check the priority. The one with higher priority will stat for another comparison with other controllers. The “failure” is eliminated out of the arbitration.

```
eq arbitrate (  < CID : Controller | txbuf : (FRAM :
   BUF) >  < CID2 : Controller | txbuf : (FRAM2 : BUF2)
   >  < BID : Bus | status : started CONF >
=
if (FRAM prior FRAM2)
then
< CID2 : Controller | > arbitrate (< CID : Controller |
   >  < BID : Bus | > CONF)
else
< CID : Controller | > arbitrate (< CID2 : Controller |
   >  < BID : Bus | > CONF)
fi .
```

A controller without messages to transfer will be removed from this arbitration directly.

```
eq arbitrate (< CID : Controller | txbuf : nilQue >  <
```

```
   BID : Bus | status : started > CONF)
=
< CID : Controller | > arbitrate (< BID : Bus | > CONF)
   .
```

In addition, we also have some other rules and functions to model the CAN bus and the CAN controller behaviors.

4.2 The Control Services

CANOpen is not only a communication protocol but also an application layer profile, so data transmission is only a part of the specification. Compared to data transmission, control services especially network and application states controls are more complex. Thus, there are more possibilities to encounter exceptions in control services. This is why our model mainly focuses on the EMCY service and the NMT service.

4.2.1 The EMCY Service

EMCY service is used to manage the application inner states. The CANOpen protocol defines two *emergency states*:

- **error free**: Error free indicates that no application internal error exists in the CANOpen device.
- **error occurred**: Error occurred indicates that there are application internal errors happened, and the errors are not resolved yet.

Figure 4.5 depicts the application internal states transition of a CANOpen device. We label each transition with a number from 0 to 5, and explain all these transition as follows:

0. After initialization, the CANOpen device enters the *error free* state if no error is detected. No EMCY object is sent out.
1. The CANOpen device detects an internal error indicated in the first three bytes of the emergency message. The CANOpen device enters into the

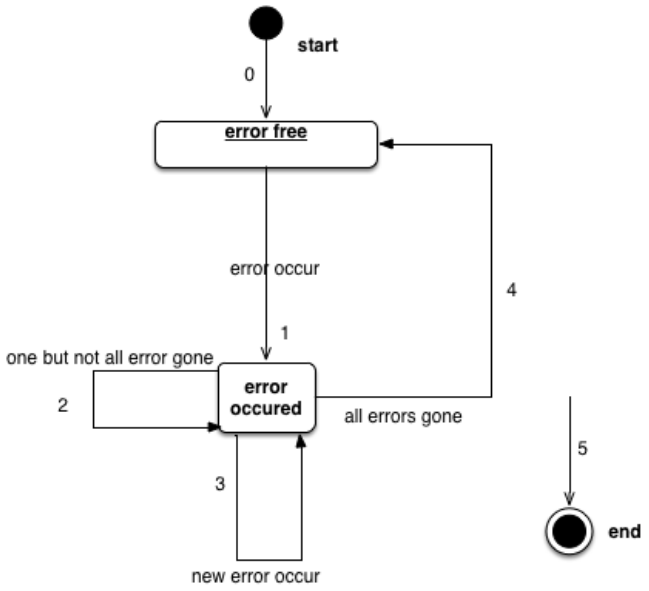

Figure 4.5: *Error States Transition*

error occurred state. An EMCY object with the appropriate error code is transmitted.

2. One of the previous errors, but not all of them, is resolved. An emergency message containing error code 0000_h(This is the *error reset* message) may be transmitted together with the remaining errors.

3. A new error occurs on the CANOpen device. The CANOpen device remains in the *error occurred* state and transmits an EMCY object with the appropriate error code. The new error code is filled in at the top of the array of error codes. It shall be guaranteed that the error codes are sorted in a timely manner.

4. All error reasons are gone. The CANOpen device switches back to the *error free* state and transmits an EMCY object with the error code *error reset / no error*.

5. CANOpen device is reset or power-off.

The rest part of this section shows how we model the EMCY protocol by Maude. Not every part of the model concerning the EMCY protocol will be listed. We model a device with EMCY service with the class as below:

```
class Node | ID : Nat, emcyRole : EmcyRole, errNo :
   Nat, errs : EmcyErrNameSet, nodeState :
   NodeStatus, emcyRecord : EmcyRecords .
```

where `ID` is the ID of the device in the CAN bus network. Since there is at most 128 nodes in a CAN system, the `ID` is natural number less than 128. Furthermore, there are not any two `Nodes` with the same ID. The attribute `emcyRole` denotes if the node is the EMCY message producer or the EMCY message consumer. The attribute `errNo` is the number of internal errors occurred and not resolved, and `errs` records all these errors. The attribute `nodeState` indicates if the node is *error free* or *error occurred*. The last attribute `emcyRecord` is for EMCY message consumer only, which records all the errors reported by EMCY message producer.

We use the following rule to model an EMCY message producer catches an application internal error and generate an EMCY message.

```
rl[nodeErrorOccur] :
< O : ErrType | validErrName : EED S >
< NodeO : Node | emcyRole : EmcyP, ID : IDN, errNo : N,
   errs : S2 >
=>
< NodeO : Node |    errNo : s(N), errs : S2 EED,
   nodeState : errorOccured >
< O : ErrType | >
(Emergency eecerr withDesc EED from IDN) .
```

In the above rule, the object `O` is an instance of class `ErrType` which defines the possible application internal error names. The node `NodeO` catches then error named `EED`, which is a constant of possible error reasons. This node generates an emergency message `(Emergency eecerr withDesc EED from IDN)` which is a subclass of `FrameData` to report its internal error. So it can be encapsulated within the class `Frame` which models a CAN bus frame. This message is also a type of `Msg` which is an pre-defined object in Full Maude. Besides the error reason `EED`, the message contains the node ID `IDN` of the producer. In

addition, the producer increases its errNO, and records the new error reason in its attribute errs. No matter what the previous EMCY state of the node is, its current state will be errorOccured.

The EMCY consumer which expects the EMCY message from node Node0 will record the necessary information when the message is delivered. The following rule models the operations of the EMCY consumer.

```
rl[emcyErrorConsumed] :
< Node0 : Node | emcyRole : EmcyC, emcyRecord : ERS >
(Emergency eecerr withDesc EED from IDN)
=>
< Node0 : Node | emcyRecord : ERS (node IDN caughtErr
   EED) > .
```

Node Node0 is the EMCY message consumer which is indicated by its attribute emcyRole. The consumer "consumes" the incoming message, and records the producer node ID IDN and error reason EED in its attribute emcyRecord.

The above rules consider the EMCY protocol independently from the underlying features of network, such as CAN controller and CAN bus. We will associate the EMCY service with the underlying features when we do the analysis in next chapter.

4.2.2 The NMT Service

NMT service defines several network states for the CANOpen device. On one hand, NMT service stipulate which communication services are supported in different network states. On the other hand, NMT provides the protocol to manipulate the states of all devices in the entire system. There are four NMT states in CANOpen, one of which can be divided into 3 sub-states. Figure 4.6 illustrates all the NMT states and the transitions between them.

- **Initialisation**: The CANOpen device enters into *initialisation* directly after power-up or reset. The NMT state *initialisation* could be divided into three NMT sub-states in order to enable a complete or partial reset of a CANOpen device.

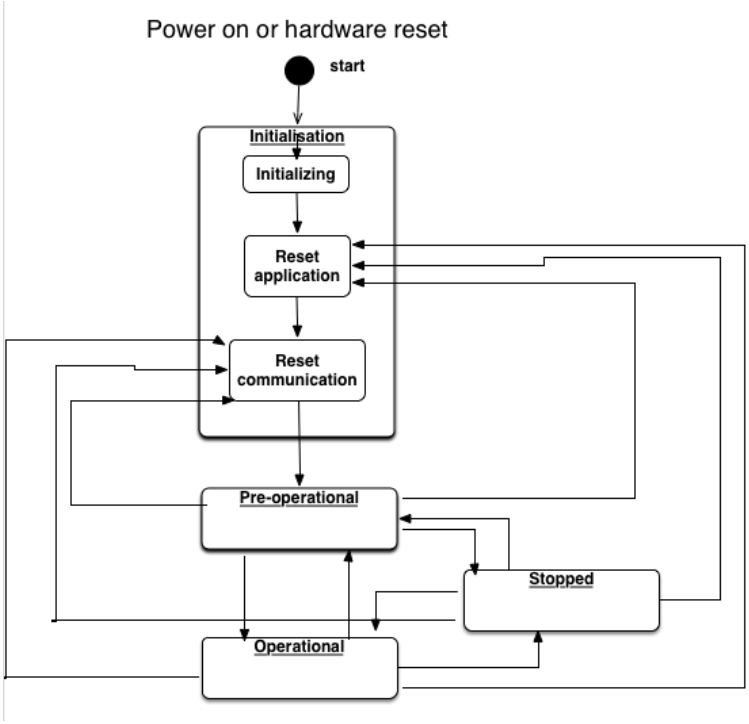

Figure 4.6: *NMT State Transition*

1. **Initialising**: This is the first NMT sub-state the CANOpen device enters after power-on or hardware reset. After finishing the basic CANOpen device initialisation the CANOpen device enters autonomously into the NMT sub-state *reset application*.

2. **Reset application**: In this NMT sub-state, the parameters of the manufacturer-specific profile area and of the standardized device profile area are set to their power-on values. After setting of the power-on values, the NMT sub-state *reset communication* is entered autonomously.

3. **Reset communication**: In this NMT sub-state, the parameters of the communication profile area are set to their power-on values. At the end of initialisation, the CANOpen device tries to transmit a specific message to the master to indicate its completion of boot-up. As soon as it is transmitted successfully, the CANOpen device switches into *pre-operational* state.

- **Pre-operational**: In the *pre-operational* state, application specific data is not allowed. Reversely, CANOpen devices can communicate by SDO objects,

so configuration data are transferred in this state. The CANOpen device may be switched into the NMT state *operational* directly under the control of the master in the network or by means of local control.

- **Operational**: In the NMT state *operational*, all communication objects are active, including the process data object. It is the normal working state for CANOpen devices.
- **Stopped**: By switching a CANOpen device into the NMT state *stopped*, it is forced to stop the communication altogether. Furthermore, this NMT state may be used to achieve certain application behavior, which is manufacture-specified. If there are EMCY messages triggered in this NMT state they are pending. The most recent active EMCY reason may be transmitted after the CANOpen device transits into another NMT state.

We use a class in Maude to model the NMT service in a CANOpen device:

```
class Node | ID : Nat, nmtState : NetworkStatus,
   nmtRole : NmtRole, slaveRecords : CommStateRecordSet,
```

where `nmtState` indicates the NMT states of this node, `nmtRole` indicates if the node is the NMT service master or the NMT servcie slaver. The last attribute `slaveRecords` is for NMT master only, which records the NMT states of all the NMT slaves.

NMT state transitions are often triggered by three reasons. The first is the hardware reset. It will cause the CANOpen device to go into NMT *initialisation* state. The second trigger is local control service initiated by application internal events. The last one is the reception of the NMT *node control service* issued by the NMT master.

Through the *node control service*, the NMT master controls the NMT state of the NMT slaves. A NMT message could be sent to one certain node or to all nodes in the CANOpen system. The NMT master controls its own NMT state machine via local services, which is not specified in the CANOpen standard CiA301. So our model does not take the NMT master's NMT state transition into consideration.

The NMT *node control service* includes the following message objects:

- **Start Remote Node**: The NMT master uses *start remote node* to change the NMT state of the selected NMT slaves into NMT state *operational*.
- **Stop Remote Node**: The NMT master uses *stop remote node* to stop all network communication of the NMT slaves except for the NMT service itself. The new state of the NMT slaves will be *stop*.
- **Enter Pre-operational**: The NMT master uses *enter pre-operational* to change the NMT state of the selected NMT slaves into *pre-operational*.
- **Reset Node**: This NMT message object can control the selected NMT slaves to reset the parameters of the manufacture-specific profile area and of the standardized device profile area. The NMT state of the slaves goes into *reset application*.
- **Reset Communication**: This NMT message object will reset the parameters of the communication profile area of all selected slaves. The NMT state of the slaves will become *reset communication*.

In addition to the *node control service*, the NMT slave will use *boot-up service* to inform the NMT master its completion of boot-up. The *boot-up service* only contains one message object.

We simply define the six message objects of *node control service* as six constants with sort `NetworkStatus`. The message object of *boot-up service* is defined by the operation:

```
sort NmtMsg .
op nodeBootUp : Oid -> NmtMsg [format (s d)] .
```

which takes the object ID of the sender as the only parameter. The sort `NmtMsg` is a subclass of `FrameData`, which means it is can be used as the payload of the class `Frame`.

Because the NMT master can control the state of the NMT slave only when the NMT slave reports its boot-up, the *boot-up service* is important in NMT service. Therefore we will show the rewriting rules of sending and receiving *boot-up service* first.

```
rl [sendBootUp] :
< Node0 : Node | ID : NID, nmtState : resetComm,
   nmtRole : slave >
=>
< Node0 : Node | nmtState : preOp >
< Node0 : Frame | id : calCOBID(1792, NID), other :
   nodeBootUp(Node0) > .

rl [receiveBootUp] :
< Node0 : Node | ID : NID, slaveRecords : CSTRecds,
   nmtRole : master >
< FID : Frame | id : COB, other : nodeBootUp(Node02) >
=>
< Node0 : Node |    slaveRecdLst : (updateRecord (node
   Node02 in preOp) toList CSTRecds) >
```

The first rule indicates that a NMT slave which is in the *reset communication* NMT state can automatically enters into the *pre-operation* state. At the same time, a *boot-up service* message is sent to report this transition. The second rule models the operations of the NMT master receiving the *boot-up service* message. The master only simply records the NMT state of the slave. There are two auxiliary functions in the above rules. One is (`op calCOBID : Nat Nat -> Nat .`) which is used to generate the proper COB-ID. Another is

```
op updateRecord_toList_ : CommStateRecord
   CommStateRecordSet -> CommStateRecordSet .
eq updateRecord NdRecd toList nilRdLt = NdRecd .
eq updateRecord (node OID in NtSt) toList ((node OID2
   in NtSt2) NdStRecdLst)
=
if OID == OID2
then
        (node OID2 in NtSt) NdStRecdLst
else
```

```
        (node OID2 in NtSt2) (updateRecord (node OID in
           NtSt) toList NdStRecdLst)
fi .
```

which will insert a new record of node ID and the related NMT state into a set if the node ID does not exists. If the node is already in the set, it will just update the node's NMT state.

The sending and receiving of NMT *node control service* is very similar, so we take the *reset application* message as an example to illustrate how we model the *node control service*.

```
crl [sendResetAppMsgToSingleNode] :
< Node0 : Node | nmtRole : master, ID : NID,
   slaveRecdLst : (CSTRecds2 (node Node02 in
   NetworkState) CSTRecds) >
=>
< Node0 : Node |    slaveRecdLst : (CSTRecds2 (node
   Node02 in resetApp) CSTRecds) >
< Node0 : Frame | id : (calCODByNodeID NID andMsgTypeID
   0), other : (changeNode Node02 toState resetApp) >
if NetworkState == stopped or NetworkState == operation
   or NetworkState == preOp .

crl [receiveResetAppMsg] :
< Node0 : Node | nmtRole : slave, nmtState :
   NetworkState >
< FID : Frame | id : NID, other : (changeNode Node0
   toState resetApp) >
=>
< Node0 : Node | nmtState : resetApp >
if NetworkState == stopped or NetworkState == operation
   or NetworkState == preOp .
```

The rule `sendResetAppMsgToSingleNode` models a NMT master sending the *reset application* message. According to the CANOpen standard, this NMT state transition is allowed when the NMT slave is in any of the three NMT states *stopped*, *pre-operation* and *operation*. This is specified by the condition of the rule. Since *node control service* is unconfirmed, once the NMT master send out this *reset application* message, it will change the state of the NMT slave in its record.

The rule `receiveResetAppMsg` defines the behaviors of a NMT slave when it receives the *reset application* message. The CANOpen standard does not describe how to deal the *reset application* message if a NMT slave is not in any of these three states *stopped*, *pre-operation* and *operation*. So we define this rule with a condition. In addition, as the standard required, the NMT slave will change its state into *reset application* if the NMT state is proper.

There are 34 rules for NMT protocols on our model. Furthermore, we also define some functions used in the rules. In next chapter, we will do the analysis based on the formal model. To the analyzing goal, we may make a little modifications of our initial model, and we will points these modifications out.

Chapter 5

Analysis of CANOpen

In this chapter, we will formally analyze the Full Maude specification of the control services in the CANOpen standard. For analysis, the first step is to specify the possbile issues. The goal of the analysis is to validate if the EMCY protocol is reliable enough to report the application internal error by the EMCY protocol and if the NMT protocol can be used to fully control the state of all NMT slaves. We want to investigate if there are inconsistencies between the EMCY consumer and the EMCY producer or the NMT slave and the NMT master, and can the inconsistencies be discovered and recovered by the CANOpen protocol. The protocol is highly nondeterministic, and consequently there is an explosion of the state database in Maude. Because of the restrict of the computing power and memory storage, it is not impossible to analysis some of the cases within a reasonable time.

With the goal of our validation, we structure the rest parts of this chapter as follows: Section 5.1 explains why the CANOpen formal model needs underlying network support for the analysis; section 5.2 illustrates how to use Maude to analyze the CANOpen protocol with the support of the CAN bus and CAN controllers; since we are interested in helping implementers and manufactures of CANOpen devices, section 5.3 shows the test cases obtained from our analysis which could be used in the testing phase of actual equipment developments.

5.1 Underlying Network Support in the Formal Model

Our analysis object is the CANOpen protocol. We have introduced that this protocol is a high-level application layer and communication profile, and it is independent of the under-layer network. So our analysis is first started on the protocol itself without consideration of the under-layer network model. Here we will use the analysis on the EMCY protocol as an example to show the simulation of CANOpen protocol.

EMCY protocol is used to report application internal errors, so before we can analyze the EMCY protocol, we have to define some of the application internal errors first. In our model, we define four constants `genericErr`, `unknownErr`, `tooHigh` and `tooLow` to represent the internal errors. With these constants, we define an initial state `init0` as follows:

```
op init0 : -> Configuration .
eq init0 = < "Producer" : Node | ID : 1, emcyRole :
   EmcyP, errNo : 0, errs : nil, nodeState : errorFree,
   emcyRecord : nilEmcyRecord >
< "Consumer" : Node | ID : 2, emcyRole : EmcyC, errNo :
   0, errs : nil, nodeState : errorFree, emcyRecord :
   nilEmcyRecord >
< "ValidErrorTypes" : ErrType | validErrName :
   (genericErr unknownErr tooHigh tooLow) > .
```

where the EMCY producer currently has no internal *error occurred* and the EMCY error record `emcyRecord` is empty.

If we use a rewrite command to check one possible state from initial state `init0`, there will not be any result given out by Maude after a very long time. This is consistent with the EMCY protocol. Because the CANOpen standard does not put a limit on the number of times the EMCY message can be sent, so the system is non-terminated. Thus, we add a new attribute `count` into the EMCY node class, which is the upper bound of the number of times an EMCY node could send an EMCY message. With this attribute, we set the upper bound of the times for the `Producer` to send EMCY messages as `6`, then we have a new initial state `init1`. We run the same *rewrite* command on the new initial state,

then we get the following result:

```
Configuration :
< "Consumer" : Node | ID : 2,count : 0, emcyRecord :
   nilEmcyRecord, emcyRole : EmcyC, errNo : 0, errs :
   nil, nodeState : errorFree >
< "Producer" : Node | ID : 1,count : 0, emcyRecord :
   nilEmcyRecord, emcyRole : EmcyP, errNo : 0,errs :
   nil, nodeState : errorFree >
< "ValidErrorTypes" : ErrType | validErrName
   :(genericErr tooHigh tooLow unknownErr)>
```

This is a possible state from the initial state `init1` where all occurred internal errors are resolved . The value `0` of the attribute `count` of `Consumer` indicates that 6 EMCY messages are sent out. Our formal model of the EMCY protocol acts well in accordance with CANOpen standard. So we will start to check if the protocol would lead the system into some unexpected status which conflicts with the protocol.

For the EMCY protocol, the CANOpen standard only depicts that the EMCY producer could report and clear application internal errors and the EMCY consumer need record the EMCY events. To check if there is any problem for the EMCY protocol, we could check the consistency between EMCY error status of the EMCY producer and the record in the EMCY consumer. With the following two search commands from initial state `init1`, we compared if there is a final state that the `errs` is not empty but `emcyRecord` is empty or `errs` is empty but `emcyRecord` is not empty.

```
(search  init1 =>! < "Producer" : Node | errs :
   ESET:EmcyErrNameSet >
< "Consumer" : Node | emcyRecord : ER:EmcyRecords >
C:Configuration such that ESET:EmcyErrNameSet =/= nil
   and ER:EmcyRecords == nilEmcyRecord . )

(search  init1 =>! < "Producer" : Node | errs :
```

```
   ESET:EmcyErrNameSet >
< "Consumer" : Node | emcyRecord : ER:EmcyRecords >
C:Configuration such that ESET:EmcyErrNameSet == nil
   and ER:EmcyRecords =/= nilEmcyRecord . )
```

Both of these two commands give the result "No solution". Thus, with our analysis result, taking no consideration of the underlying network, the EMCY protocol works well. Obviously, this is far from what we expect. We hope we could find some issues about this protocol by the Maude model. They are probably some small issues because CANOpen protocol has been put into practice for decades. So this is one of the reasons why we think about taking underlying network into our model.

Another reason is because the CANOpen protocol implicitly assumes the underlying network protocol is reliable, so it does not provide the verification mechanism. The reliability contains two aspects: one is that the message sent out by CANOpen device will always be delivered; the other is that the messages deliver order is the same as the messages sending order. But in the real case, it does not sound reasonable that a message is successfully delivered when the hard link is cut off. This point is also mentioned in [20] where they believe message lost could happen in the CAN controllers.

In the embedded automation system, the CANOpen protocol is widely used with CAN bus. More precisely, CAN bus was introduced first, then CANOpen merged later which was designed as a *CAN Application Layer(CAL)* protocol. So we have strong reasons to use the features of the CAN bus and CAN controllers as the lower level network prototype.

5.2 Analysis of CANOpen with Low-level Network Feature

This section will show some of our analysis results of the CANOpen based on our formal model in chapter 4. The goal of the analysis to find if there is any weakness in CANOpen, so the analysis in this section mainly focuses on the erroneous behaviors of EMCY protocol and NMT protocol.

5.2.1 Complements to The Formal Model for Analysis

The state transitions in CANOpen is complicated and there are so many possibilities that one state to another that bring difficult to our analysis. Thus, we add additional attributes `failedRxbuf` and textttfailedTxbuf to class `Controller` in our model. These two attributes are supposed to capture the discarded incoming messages and outgoing messages respectively for a CANOpen device. In addition, CANOpen controllers will discard messages when the buffer is full in our model, but we cannot make sure if the message is discarded or just not sent if we do not have these two auxiliary attributes. They are very helpful to locate issues of CANOpen protocol in our model.

To integrate the `Controller` and the `Bus` with the CANOpen `Node`, we also need some other functions and attributes. Since each CANOpen device has one and only one CAN controller associated with it, we add another attribute `controllerID` to class `Node` whose value is the ID of an instance of `Controller`. Furthermore, we also define a function (`op pushMsg_toController_ : Object Object -> Object .`) which pushes a CAN bus `Frame` to a `Controller`.

5.2.2 Simulation of the EMCY Protocol

The goal EMCY protocol is to report application internal errors to other nodes in the CANOpen system. The device behaviors for the producer after its sending the EMCY message and the device behaviors of the receiver after its receiving the message are not part of the CANOpen standard. So what we need to analysis is that if the EMCY messages can always be delivered successfully.

Before simulation, we define a initial state `init2` first as follows:

```
op init2 : -> Configuration .
eq init2 =
< "Sensor" : Node |   ID : 1, emcyRole : EmcyP,
    errNo : 0, errs : nil, count : 4,
    nodeState : errorFree, emcyRecord : nilEmcyRecord,
    controllerID : "S_Controller" >
< "Center" : Node |   ID : 2, emcyRole : EmcyC,
    errNo : 0, errs : nil, count : 0,
```

```
    nodeState : errorFree, emcyRecord : nilEmcyRecord,
    controllerID : "C_Controller" >
< "ValidErrorTypes" : ErrType |
    validErrName : (tooHigh tooLow) >
GLOBAL (
    < "C_Controller" : Controller |
        txbuf : nilQue, rxbuf : nilQue, filter : 81,
        txbufCap : 4, rxbufCap : 4,
        failedTxbuf : nilQue, failedRxbuf : nilQue >
    < "S_Controller" : Controller |
        txbuf : nilQue, rxbuf : nilQue, filter : 82,
        txbufCap : 4, rxbufCap : 4,
        failedTxbuf : nilQue, failedRxbuf : nilQue >
    < "Bus" : Bus | status : free, owner : nilOid >) .
```

where we set the size of all buffers withing all nodes including both sending and receiving buffers as 4. Similar to analysis in section 5.1, we also need our self-defined internal error names which is listed in `ValidErrorTypes`. But to reduce the analysis state space, we only define two error names in the initial state `init`. We assign the message type of the EMCY message an number `80`, so according to this value and the device ID, we also configure the values of attribute `filter` as `81` and `82` for `C_Controller` and `S_Controller` respectively.

To find if an EMCY message can get lost, we can use a similar *search* command in section 5.1

```
(search [1]  init2 =>!
< "Sensor" : Node | errs : ESET:EmcyErrNameSet >
< "Center" : Node | emcyRecord : ER:EmcyRecords >
C:Configuration such that ESET:EmcyErrNameSet =/= nil
and ER:EmcyRecords == nilEmcyRecord . )

(search [1] init2 =>!
< "Sensor" : Node | errs : ESET:EmcyErrNameSet >
< "Center" : Node | emcyRecord : ER:EmcyRecords >
C:Configuration such that ESET:EmcyErrNameSet == nil
```

```
and ER:EmcyRecords =/= nilEmcyRecord . )
```

to check if there is at least one final state where the `Sensor` occurred error record is not consistent with the EMCY record in `Center`. After a couple of seconds, Maude returns "No solution". This is reasonable result because we set the size of buffer as 4 and the value of `count` is also 4. It means there will not be message lost under this condition.

To expose the potential problem, we change the size of buffers to 1 and keep the `count` value as 4. We do the same *search* simulation. This time Maude gives out a solution:

```
C:Configuration -->
GLOBAL (
    < "Bus" : Bus | owner : nilOid,status : free >
    < "C_Controller" : Controller |
        failedRxbuf : nilQue, failedTxbuf : nilQue,
        filter : 81, rxbufCap : 1,
        rxbuf : nilQue, txbufCap : 1, txbuf : nilQue >
        < "S_Controller" : Controller |
        failedRxbuf : nilQue,
        failedTxbuf :(< "Sensor" : Frame | id : 81,
                other : Emergency eecerr withDesc
                   tooHigh from 1 > :
            < "Sensor" : Frame | id : 81,
                other : Emergency eecerr withDesc
                   tooHigh from 1 >),
            filter : 82, rxbufCap : 1, rxbuf : nilQue,
            txbufCap : 1, txbuf : nilQue >)
    < "ValidErrorTypes" : ErrType |
        validErrName :(tooHigh tooLow)> ;
    ER:EmcyRecords --> nilEmcyRecord ;
    ESET:EmcyErrNameSet --> tooHigh tooHigh ;
    V#0:Node --> Node ;
    V#1:AttributeSet --> ID : 1,
    controllerID : "S_Controller", count : 0,
```

```
emcyRecord : nilEmcyRecord,
emcyRole : EmcyP, errNo : 2,
nodeState : errorOccured ;
V#2:Node --> Node ;
V#3:AttributeSet --> ID : 2,
controllerID : "C_Controller", count : 0,
emcyRole : EmcyC, errNo : 0,
errs : nil,nodeState : errorFree
```

where there are two internal errors `tooHigh` in `Sensor` but no record about these two errors in `Controller`. This means EMCY messages reporting these two errors are lost. This is confirmed by the value of `failedTxbuf` in `S_Controller`. It is possible that the EMCY messages are discarded by the controller of `Sensor` since the `txbuf` could be full. So with the model of CAN bus and CAN controller, the EMCY protocol cannot ensure the EMCY message is delivered.

This could lead some critical problems in real case. For example, a warehouse requires the temperature stays between -20 to -15 degree. A sensor inside detects the temperature of the warehouse. If the temperature exceeds the temperature range, the sensor could report this to the central control unit. The central control unit will start or stop the cooler accordingly. It is possible that the cooler encounters some internal error and report this to the central control unit. If this message get lost, the control unit will never know the cooler cannot work. So the temperature of the warehouse may be over the upper bound. This would be a serious problem.

5.2.3 Simulation of the NMT Protocol

The purpose of the NMT protocol is to manage the network state of the CANOpen device. The CANOpen standard depicts how to use the NMT protocol to control CANOpen device states, but the reasons triggering the control service are not specified. On the other side, there is one master in the CANOpen system recording all other slaves' network states, so our analysis can only focus on the consistency of network states between NMT slave and the record of NMT master.

First, we will use a very simple *search* simulation to show the basic behaviors

of the NMT protocol. For a new CANOpen device to join into a CANOpen system, if it can enter into a normal working state. We define the initial state `init3` as follows:

```
op init3 : -> Configuration .
eq init3 =
GLOBAL (
    < "BUS" : Bus | status : free, owner : nilOid >
    < "CONTROLLER_ONE" : Controller |   txbuf : nilQue,
        rxbuf : nilQue, filter : (20000 21792),
        txbufCap : 100, rxbufCap : 100,
        failedRxbuf : nilQue, failedTxbuf : nilQue >
    < "CONTROLLER_TWO" : Controller |   txbuf : nilQue,
        rxbuf : nilQue, filter : (10000),
        txbufCap : 100, rxbufCap : 100,
        failedRxbuf : nilQue, failedTxbuf : nilQue >)
< "Node_ONE" : Node | ID : 1, errNo : 0,
        nodeState : errorFree, nmtRole : master,
        controllerID : "CONTROLLER_ONE",
        slaveRecdLst : nilRdLt,
        networkState : operation >
< "Node_TWO" : Node | ID : 2, errNo : 0,
        nodeState : errorFree, nmtRole : slave,
        controllerID : "CONTROLLER_TWO",
        slaveRecdLst : nilRdLt,
        networkState : initialising > .
```

where node `Node_ONE` acts as the NMT master and node `Node_TWO` acts as the NMT slave. Similar to EMCY analysis, we configure controller `CONTROLLER_ONE` associated with node `Node_ONE` and controller `CONTROLLER_TWO` associated with node `Node_TWO`. There two types of message in this simulation. We assign the message type number 1792 to the boot-up message and 0 to the NMT message, which is because NMT message has higher priority. We also need to configure the attribute `filter` to accept these two set of messages. The NMT slave is in NMT state `initialising` which means it is in the boot-up process. On the

other, the attribute slaveRecdLst in the NMT master is empty, so the NMT slave is new to join the CANOpen system. To validate if the NMT slave can work properly, we need to find if the NMT slave can enter into operation state. We execute the *search* command as below and get the result from Maude:

```
(search [1] init3 =>* C:Configuration
    < "Node_TWO" : Node | networkState : operation > .)

Solution 1
C:Configuration -->
GLOBAL (
    < "BUS" : Bus | owner : nilOid, status :
       transmitted >
    < "CONTROLLER_ONE" : Controller |
        failedRxbuf : nilQue, failedTxbuf : nilQue,
        filter :(20000 21792), rxbufCap : 100,
        rxbuf : nilQue, txbufCap : 100 ,txbuf : nilQue >
    < "CONTROLLER_TWO" : Controller |
        failedRxbuf : nilQue, failedTxbuf : nilQue,
        filter :(1001 10000), rxbufCap : 100,
        rxbuf : nilQue, txbufCap : 100, txbuf : nilQue>)
< "Node_ONE" : Node | ID : 1,
    controllerID : "CONTROLLER_ONE", errNo : 0,
    networkState : operation, nmtRole : master,
    nodeState : errorFree,
    slaveRecdLst : node "Node_TWO" in operation > ;
V#0:Node --> Node ;
V#1:AttributeSet --> ID : 2,
    controllerID : "CONTROLLER_TWO", errNo : 0,
    nmtRole : slave, nodeState : errorFree,
    slaveRecdLst : nilRdLt
```

One solution is obtained from Maude where node Node_TWO is in operation state and the attribute slaveRecdLst in node Node_ONE also records this state correctly. So for a new CANOpen device to join into a CANOpen system, the

new device could work properly.

However, we are also interested in if a new CANOpen device can always join into a CANOpen system. To analyze this problem, we reduce the size of buffers to 1. This could help us to find the a counter example more quickly. Because decreasing the size of buffers could reduce the rewriting steps to the state where the buffer is full. In addition, we need a new initial state which introduce another CANOpen node in our system. We name this new initial state as `init4`. Compared to `init3`, the added node has the value of attribute `ID` 3 and the object ID `Node_THREE`. Moreover, there must be a corresponding instance of `Controller` in `init3`, and the `filter` in each `Controller` should be correct configured. With this initial state, we try to find a state that the new CANOpen device has finishing boot-up and sent out the boot-up message to inform the NMT master, but the message cannot be delivered. According to the CANOpen protocol, the message will only be sent once and no acknowledge is required, so we believe that the NMT master will never know the exist of the new NMT slave. We have the following *search* command, and we also get the result from Maude:

```
(search [1] init4 =>* C:Configuration
    < "Node_TWO" : Node | networkState : preOp >
    GLOBAL (C2:Configuration
        < "CONTROLLER_ONE" : Controller |
            failedRxbuf : T:Que > )
    such that T:Que ==
        < "CONTROLLER_TWO" : Frame | id : 21792,
        other : node "Node_TWO" bootUp > .)

Solution 1
C2:Configuration -->
    < "BUS" : Bus | owner : nilOid,
        status : transmitted >
    < "CONTROLLER_THREE" : Controller |
        failedRxbuf : nilQue, failedTxbuf : nilQue,
        filter : 10000, rxbufCap : 1, rxbuf : nilQue,
        txbufCap : 1, txbuf : nilQue >
    < "CONTROLLER_TWO" : Controller |
```

```
        failedRxbuf : nilQue, failedTxbuf : nilQue,
        filter : 10000, rxbufCap : 1, rxbuf : nilQue,
        txbufCap : 1, txbuf : nilQue > ;
C:Configuration -->
    < "Node_ONE" : Node | ID : 1,
        controllerID : "CONTROLLER_ONE",
        errNo : 0, networkState : operation,
        nmtRole : master, nodeState : errorFree,
        slaveRecdLst : nilRdLt >
    < "Node_THREE" : Node | ID : 3,
        controllerID : "CONTROLLER_THREE",
        errNo : 0, networkState : preOp,
        nmtRole : slave, nodeState : errorFree,
        slaveRecdLst : nilRdLt > ;
T:Que -->
    < "CONTROLLER_TWO" : Frame |
        id : 21792,other : node "Node_TWO" bootUp > ;
V#0:Node --> Node ;
V#1:AttributeSet -->
    ID : 2, controllerID : "CONTROLLER_TWO", errNo : 0,
    nmtRole : slave, nodeState : errorFree,
    slaveRecdLst : nilRdLt ;
V#2:Controller -->
    Controller ;
V#3:AttributeSet -->
    failedTxbuf : nilQue,
    filter :(20000 21792 30000 31792),
    rxbufCap : 1, rxbuf :
        < "CONTROLLER_THREE" : Frame | id : 31792,
            other : node "Node_THREE" bootUp >,
    txbufCap :1, txbuf : nilQue
```

Since the boot-up message probably is delayed in transmission (CAN bus busy), it is much easier to describe the search by the attribute `failedRxbuf` in `CONTROLLER_ONE` which is the NMT master. If node `Node_TWO` is in NMT state `preOp` and its boot-up message can be found in the `failedRxbuf` of

CONTROLLER_ONE, then we can confirm that node Node_TWO joins into this system failed. This condition do not only happen when a new device needs to join the CANOpen system, but also could happen when a NMT Master send *reset communication* and *reset node* message to NMT slave.

In the CANOpen standard CiA301, it only specifies the expected NMT states transition from one state to another. However, it does not indicate the default behaviors when a device receives a NMT service that does not match any satisfied states transition condition. For example, a NMT slave is currently in operation state when it receives an NMT message *enter pre-operation* from a NMT master. So another interesting property is that, if this uncovered circumstance could happen. To validate this property, we define another initial state init5:

```
op init5 : -> Configuration .
eq init5 =
GLOBAL (
    < "BUS" : Bus | status : free, owner : nilOid >
    < "CONTROLLER_ONE" : Controller | txbuf : nilQue,
        rxbuf : nilQue, filter : (20000 21792),
        txbufCap : 100, rxbufCap : 1,
        failedRxbuf : nilQue, failedTxbuf : nilQue >
    < "CONTROLLER_TWO" : Controller | txbuf : nilQue,
        rxbuf : nilQue, filter : (10000), txbufCap : 1,
        rxbufCap : 1, failedRxbuf : nilQue,
        failedTxbuf : nilQue >)
< "Node_ONE" : Node | ID : 1, errNo : 0,
    nodeState : errorFree,
    controllerID : "CONTROLLER_ONE",
    nmtRole : master,
    slaveRecdLst : node "Node_TWO" in operation,
    networkState : operation >
< "Node_TWO" : Node | ID : 2, errNo : 0,
    nodeState : errorFree,
    controllerID : "CONTROLLER_TWO",
    nmtRole : slave, slaveRecdLst : nilRdLt,
    networkState : operation > .
```

where node `Node_ONE` is the NMT master and node `Node_TWO` is the NMT slave. The NMT slave is in NMT state `operation` which is correctly recorded in the NMT master. We use the following *search* command to find a state that the NMT slave is in NMT state `operation` but its related `rxbuf` contains a NMT service *Start Remote Node*. After a few minutes, we get a solution from Maude. Thus, it is possible that a CANOpen system reaches to an unspecified network state. This is another potential problem of the NMT protocol.

```
(search [1] init3 =>* C:Configuration
< "Node_TWO" : Node | networkState : operation >
GLOBAL (
    C2:Configuration
    < "CONTROLLER_TWO" : Controller |
        rxbuf : < "CONTROLLER_ONE" : Frame |
            id : 10000,
            other : (changeNode "Node_TWO" toState
               operation) > >) .)

Solution 1
C2:Configuration -->
    < "BUS" : Bus | owner : nilOid,
        status : transmitted >
    < "CONTROLLER_ONE" : Controller |
        failedRxbuf : nilQue, failedTxbuf : nilQue,
        filter :(20000 21792), rxbufCap : 1,
        rxbuf : nilQue, txbufCap : 100,
        txbuf : nilQue > ;
C:Configuration -->
    < "Node_ONE" : Node | ID : 1,
        controllerID : "CONTROLLER_ONE", errNo : 0,
        networkState : operation, nmtRole : master,
        nodeState : errorFree,
        slaveRecdLst : node "Node_TWO" in operation > ;
V#0:Node --> Node ;
V#1:AttributeSet --> ID : 2,
    controllerID : "CONTROLLER_TWO", errNo : 0,
```

```
    nmtRole : slave, nodeState : errorFree,
    slaveRecdLst : nilRdLt ;
V#2:Controller --> Controller ;
V#3:AttributeSet -->
    failedRxbuf : < "CONTROLLER_ONE" : Frame |
        id : 10000,
        other : changeNode "Node_TWO" toState preOp >,
    failedTxbuf : nilQue, filter : 10000, rxbufCap : 1,
    txbufCap : 1, txbuf : nilQue ;
V#4:Frame --> Frame ;
V#5:AttributeSet --> (none).AttributeSet
```

We also find some other problems when building the model. These problems are caused by the underspecification of CANOpen protocol. For example, in CiA301, it mentions a CANOpen device can switch into NMT state *operational* also by means of local control but without any more explanation. This conflicts with the NMT service *start remote node*. Because of its logical basis and its initial model semantics, a Maude module defines a precise mathematical model[21]. When building the model by Maude, we can find the underspecification from the documentations and specifications.

With consideration of low-level network features, we find some potential issues in the control service of CANOpen protocol. Our model and analysis are based on CANOpen standard. Because of the complexity of the CANOpen protocol and the limitation of time, we also need some confirmation from DNV GL on whether our analysis is valuable. They showed our analysis results to FMC Technology which is an American global provider that provides equipment and services for subsea system. FMC Technologies' subsea systems business encompasses a wide range of equipment and technologies that are required to explore, drill and develop offshore oil and gas fields. They have a strong global presence in all of the world's major basins. With reference of their industry experience, they give us some positive feedbacks that they find the analysis realistic and acknowledge that these problems potentially exists in the CANOpen protocol. They cannot completely rely on the CANOpen standard only, and some other manufacture-specific process are required to handle these problems. Hence, their comments are also supportive to our analysis results.

5.3 Examples of Test Case

The motivation of our modeling and analysis is from the industry. We hope our work could also be helpful for implementers and manufacturers of CANOpen devices. Most of them probably have no background of Maude or even formal analysis. The test case is a direct way for the engineers and developers to understand and practice. Actually, in the development of subsea equipment, testing-driven model is widely used, which is supposed to help developers to derive test cases from it. After running them, test results can be evaluated automatically. The formal model helps to check the design and requirements before implementation as well[22].

From our abstract model and the analysis applied to the model, we can obtain specific abstract test cases. We list some test case examples by Table 5.1, Table 5.2 and Table 5.3, which are related to the issues we discovered in CiA301. If the equipments using EMCY service for internal error reporting, test case described by Table 5.1 can test if errors can be reliably delivered. Table 5.2 refers to the condition discovered by our analysis that a new NMT slave probably could not join a CANOpen system. The test case in Table 5.3 is to test if the system can handle unexpected NMT states not explicitly depicted in CiA301.

Table 5.1: Test Case Example of Reporting Error By EMCY Service

Test Name	utilization of EMCY service to report internal error
Test Description	In the system, there is at least one CANOpen device as the EMCY producer and one CANOpen device as the EMCY consumer. The EMCY producer will use EMCY service to report internal errors, and the EMCY consumer can record the error.
Precondition	1. Both the EMCY producer and the EMCY consumers are in the same CANOpen system 2. At first, the EMCY producer is in *error free* state and no error record of the EMCY producer in the EMCY consumer 3. An internal error incurred, the EMCY producer sends out EMCY message
Expected Result	1. The EMCY producer is in the *error occurred* state 2. The EMCY consumer records the error

These abstract test cases can be concretized to practical test cases for system under tests(SUT). One way is for example by attribute translating from the abstract test case to concrete level. The concrete stimulation can then be

Table 5.2: Test Case Example of adding new CANOpen device

Test Name	add new device into CANOpen system
Test Description	A CANOpen device can join a CANOpen system which means the new device can communicate with other devices in the network. The new device is a NMT slave, and it will in the management of the NMT master in the CANOpen system.
Precondition	1. The NMT master of an existing CANOpen system is in *operation* state. 2. A new CANOpen device is plugged into the existing CANOpen system, and no record of the new device in the NMT master. 3. The new CANOpen is powered up.
Expected Result	1. The NMT slave is in *operation* state. 2. The NMT master records that the NMT slave is in *operation*.

Table 5.3: Test Case Example of Controlling NMT Slaves

Test Name	control CANOpen device network state
Test Description	In CANOpen system, a NMT master can record and control the state of NMT slaves. There is at least a NMT slave in the CANOpen system.
Precondition	1. The NMT master of an existing CANOpen system is in *operation* state. 2. The NMT slave is in *operation* state. 3. The NMT master records the state of the NMT slave as *operation*. 4. The NMT master sends *Enter Pre-operational* message to the NMT slave
Expected Result	1. The NMT slave enter into *pre-operation* state. 2. The NMT master records that the NMT slave is in *pre-operation*.

performed in the SUT by the test platform. The output of the SUT can also be generalized to the abstract level of the test model, and be compare to the expected outputs encoded in the abstract test cases. So test evaluation is based on a abstract/concreteness relation between the traces of the abstract test module and the concrete system under test.

Part III

Conclusion

Chapter 6

Concluding and Future Works

6.1 Contributions and Study Results

Actually, CANOpen is a complex protocol that consists of many services. The protocol description covers many aspects, from very low level details (frames, bytes, bits etc) to very high level explanations. The services (NMT, EMCY, etc) are also interconnected, making it quite complicated. Moreover, there are also options and parameters which gives the protocol a large configuration space. The protocol is also highly dependent on the applications. Without a complete picture of the CANOpen protocol and underlying network prototype, it is easy to end up, for example with a model that over-approximate the behaviors and hence far from the actual protocol behaviors, and we may get the fals-positive analysis results.

This thesis models parts of the CANOpen protocol and related underlying network features, then investigates the soundness of the CANOpen protocol. In particular, it has investigated the EMCY protocol and the NMT protocol by building and analyzing the respective Maude formal model. CANOpen is designed for industry utilization and Maude is a formal tool based on mathematical theory, so this thesis and our work are also a combination of formal method with industry application.

In chapter 1, we proposed several problems to be addressed in this thesis. We will give a concluding answers to these problems according to the work showed in the thesis.

1. How can we use Maude and rewriting logic to model a complex

communication protocol such as CANOpen?

To model a communication protocol, we need to have a deep understanding of the protocol first. The next step is to find the "interesting" parts of the protocol where potential issues could occur. Because of the complexity and the time constraints, we cannot model every part of the protocol. With the understanding of the protocol, we can have ideas of the possible vulnerabilities. Moreover, we also determine our research range with reference to practical experiences from DNV GL. In the other hand, choosing the appropriate modeling level is important as well. A model with every detail of a real system typically cannot execute efficiently, so it is not useful for analysis and validation. Actually, a model is always an abstract of a real system omitting some details but not too much. A communication system is a collection of components (nodes, messages, etc), can then be represented as a multi-set of objects. Maude and its extension Full Maude support object-oriented specifications by defining *class*, *object*, *message* and the operation of message passing. The dynamic behaviors of CANOpen can be directly modeled by rules in a more object-oriented style.

2. What are the benefits of formal analysis when applied to the CANOpen protocol?

Before analysis, we need build the formal model first. CANOpen protocol is depicted by human languages with ambiguity and underspecification. But formal model is precise, it can help us to find ambiguous and underspecification problems. A typical testing on CANOpen devices is black box, ones cannot locate errors easily when finding the problems. The formal analysis can expose the details of states in a system, even the internal states. This is helpful to find the reasons of found problems. The execution of our formal analysis tool Maude is efficient, and it also provides different analysis methods (*frew*, *search* etc) which provides flexibility to validate different properties of CANOpen. Though our model and analysis on the model, we find that EMCY service is not always reliable, a NMT slave cannot be ensured to join into a CANOpen system successfully and some other problems. Hence, formal model tools and analysis such as Maude are useful to validate communication protocol like CANOpen.

3. Considering validations, can issues discovered by the analysis be understood by the application engineer? Are the issues found relevant and realistic for equipments suppliers using CANOpen?

We find some issues when doing analysis on CANOpen. But we also need

to check if the issues are valuable in the industry production, so we provide our analysis results to both DNV GL researchers and FMC engineers. DNV GL processes verification and certifications of subsea systems or equipment around the world, and FMC as the manufacture provides industry-leading subsea equipments and services. Their positive feedbacks confirm that issues discovered by our analysis could actually exist in realistic case. It needs specific complements to the CANOpen protocol when implementing CANOpen equipments.

4.How to identify the discovered issues on real system?

Finding the potential issues by formal analysis is the first step. DNV GL and we hope these discovered issues can be helpful in their system testing and verification. This is also our initial motivation of this master project. We can translate our formal analysis into testing cases: the initial state of formal analysis is seen as precondition in testing; the analysis result is described as expected or unexpected testing result. In the other side, the potential issues can also be taken into consideration by developer who develop and implement the subsea facilities. The equipments could be more robust if the developers could handle the problems in the design and development procedure.

Because of the time constraint, our model and analysis also have shortcomings. Maude is very efficient, and it can be comparable with the state-of-the-art tool SPIN[23]. When we are doing rewriting from a complex initial state, we sometimes cannot obtain a result from Maude. So we have to improve the efficiency of our formal model. One way is to ignore more details which will not affect the analysis results. Another way is to refine our model's expression pattern. This requires more experiences and skills in Maude model language.

6.2 Future Works

It is always desirable to explore if there is any more issues existing in CANOpen protocol. We have three suggestions which may be helpful in this direction. The first is real-time consideration. Since CANOpen is real-time communication, every message should be delivered within a fixed time interval. If we could take the time factor into consideration, it may expose more issues. The second suggestion is to study a real case. CANOpen is highly dependent on applications, the standard leaves many options and flexibilities for the specific applications. With a real case, we can model more behaviors which can be seen

as the complements of CANOpen. Finally, we may apply some other analysis methods on the formal model. Besides the *frew* and *search*, Maude also supports other analysis methods, such as *linear temporal logic model checking* which also focuses on reachability property. *Linear temporal logic* allows specification of properties such as safety properties and liveness properties. We can check more properties of the CANOpen system, so there are more chances to discover other potential issues.

Appendices

Appendix A

Maude Code for CANOpen Model

```
fmod PARA-QUE{X :: TRIV} is
    protecting NAT .

    sorts NeQue Que .
    subsort X$Elt < NeQue < Que .

    ****Define ground term of Queue
    op nilQue : -> Que [ctor] .

    ****Queue constructor
    op _:_ : Que Que -> Que [ctor assoc id: nilQue] .

    ****Non-empty Queue Contructor
    op _:_ : NeQue Que -> NeQue [ctor ditto] .
    op _:_ : Que NeQue -> NeQue [ctor ditto] .

    vars E E2 : X$Elt .
    vars Q Q2 : Que .
    vars NEQ NEQ2 : NeQue .

    ****Add element to the end of queue
    op _addTo_ : X$Elt Que -> Que .
****    eq E addTo nilQue = Que : E .
```

```
    eq E addTo Q = Q : E .

    ****Eliminate the element from the head of queue
    op pop : Que -> Que .
    eq pop(nilQue) = nilQue .
    eq pop(E) = nilQue .
    eq pop(E : NEQ) = NEQ .

    ****Get the first element of Queue
    op top : Que -> X$Elt .
    eq top(E : NEQ) = E .

    ****Length of a Queue
    op length : Que -> Nat .
    eq length(nilQue) = 0 .
    eq length(E : Q) = 1 + length(Q) .
endfm

(omod MESSAGE is
    including NAT .

    sort CanId .
    op 0 : -> CanId .       ***** NMT service
    op 1792 : -> CanId .    ***** NMT error control
    op 1408 : -> CanId .    ***** SDO tx
    op 1536 : -> CanId .    ***** SDO rx

    sort FrameData .

    class Frame |   id : Nat,
                    other : FrameData .

    vars FID FID2 : Oid .
    vars N N2 : Nat .

    op _prior_ : Object Object -> Bool .
    eq (< FID : Frame | id : N >) prior (< FID2 : Frame |
     id : N2 >) = N < N2 .
```

```
    sort MsgContent .
    subsort FrameData < MsgContent .

    msg _withID_from_ : MsgContent Nat Oid -> Msg .
endom)

(fmod BUS-STATUS is
    sort BusStatus .

    op free : -> BusStatus .
    op started : -> BusStatus .
    op arbitrated : -> BusStatus .
    op transmitting : -> BusStatus .
    op transmitted : -> BusStatus .
endfm)

(omod BUS is
    including BUS-STATUS .

    class Bus | status : BusStatus,
                owner : Oid .
endom)

(omod MSG-RULES is
    including MESSAGE .
    including BUS .
    including CONTROLLER .
    including PARA-QUE{QUEUE-ELEMENT} .

    sort GlobalConfiguration .
    subsort GlobalConfiguration < Object .
    op GLOBAL : Configuration -> GlobalConfiguration [ctor] .

    sort NilOid .
    subsort NilOid < Oid .
    op nilOid : -> NilOid [ctor] .
```

```
var BID : Oid .
var BSStatus : BusStatus .
vars CID CID2 : Oid .
vars FRAM FRAM2 : Object .
vars BUF BUF2 : Que .
vars NEBUF NEBUF2 : NeQue .
var CONF : Configuration .
var DATA : FrameData .
var COD : Nat .
var FILTER : NatSet .
var N : Nat .
var FID : Oid .

**** opeartion defination
op arbitrate : Configuration -> Configuration .
**** if the bus status is not "started", then ignore this
**** operation
ceq arbitrate (< BID : Bus | status : BSStatus > CONF) =
    < BID : Bus | > CONF if BSStatus =/= started .
**** if there is no message to send, controller will
**** quit the competition
eq arbitrate (< CID : Controller | txbuf : nilQue >
 < BID : Bus | status : started > CONF) =
    < CID : Controller | >
    arbitrate (< BID : Bus | > CONF) .
**** if there are 2 or more controllers in the
**** configuration wanna send message,
**** compare each two of them to find the highest
****message_id priority
eq arbitrate ( < CID : Controller | txbuf :
 (FRAM : BUF) >
 < CID2 : Controller | txbuf : (FRAM2 : BUF2) >
 < BID : Bus | status : started >
 CONF )
    =
   if (FRAM prior FRAM2)
        then < CID2 : Controller | > arbitrate (
        < CID : Controller | > < BID : Bus | > CONF)
```

```
   else
        < CID : Controller | > arbitrate (
        < CID2 : Controller | > < BID : Bus | > CONF)
   fi .
**** if there is only one controller which has message
**** to send, then it is the winner.
eq arbitrate ( < CID : Controller | txbuf : NEBUF >
               < BID : Bus | status : started > )
    =
   < CID : Controller | > < BID : Bus | status : arbitrated,
    owner : CID > .

**** operation defination
op deliver_to_ : Msg Configuration -> Configuration .
eq deliver (DATA withID COD from CID) to
 < CID2 : Controller | rxbuf : BUF, filter : FILTER,
 rxbufCap : N, failedRxbuf : BUF2 > =
    if CID == CID2
        then
            < CID2 : Controller | >
    else
        if ((FILTER percolate COD) and
           (length (BUF)) < N)
            then
                < CID2 : Controller | rxbuf : (BUF :
                < CID : Frame | id : COD, other : DATA >) >
        else
            if FILTER percolate COD ****rxbuf overflows
            then
                < CID2 : Controller | failedRxbuf : (BUF2 :
                < CID : Frame | id : COD, other : DATA >) >
            else
                < CID2 : Controller | >
            fi
        fi
    fi .

**** operation defination
```

```
op broadcast_in_ : Msg Configuration -> Configuration .
**** Ignore if the controller is the sender
eq broadcast (DATA withID COD from CID) in
    (< CID : Controller | > CONF) =
    < CID : Controller | >
    (broadcast (DATA withID COD from CID) in CONF) .
**** change the bus status if frame is
**** delivered to all controllers
eq broadcast (DATA withID COD from CID) in
    ( < BID : Bus | status : transmitting, owner : CID > ) =
    < BID : Bus | status : transmitted, owner : nilOid > .
**** deliver the frame to one controller which is
**** different from the sender
ceq broadcast (DATA withID COD from CID) in
      ( < CID2 : Controller | > < BID : Bus | owner : CID,
                        status : transmitting > CONF ) =
    (deliver (DATA withID COD from CID) to
    < CID2 : Controller | >)
    broadcast (DATA withID COD from CID) in
    ( CONF < BID : Bus | >)
    if CID =/= CID2 .

rl[startArbitration] :
    GLOBAL(< BID : Bus |    owner : nilOid,
                    status : free >
    < CID : Controller | txbuf : NEBUF >
    CONF)
    =>
    GLOBAL( arbitrate (
    < BID : Bus |   status : started >
    < CID : Controller | >
    CONF) ) .

rl[transmitMsg] :
    GLOBAL (
    < CID : Controller | txbuf : ( < FID : Frame |
      id : COD, other : DATA > : BUF) >
    < BID : Bus | owner : CID, status : arbitrated >
```

```
        CONF )
        =>
        GLOBAL (
        < CID : Controller | txbuf : BUF >
        broadcast (DATA withID COD from CID) in (
        < BID : Bus | status : transmitting >
        CONF) ) .

    rl[backToIdle] :
        GLOBAL (
        < BID : Bus | owner : nilOid, status : transmitted >
        CONF)
        =>
        GLOBAL (< BID : Bus | status : free >
        CONF ) .
endom)

(omod EMERGENCY-OBJ is
    sort EmcyErrCode .
    op eecerr : -> EmcyErrCode [ctor] .
    op eecclear : -> EmcyErrCode [ctor] .

    sort EmcyErrBody .
    op clearErr : -> EmcyErrBody [ctor] .
    op genericErr : -> EmcyErrBody [ctor] .
    op unknownErr : -> EmcyErrBody [ctor] .
    op tooHigh : -> EmcyErrBody [ctor] .
    op tooLow : -> EmcyErrBody [ctor] .

    sort EmcyErrNameSet .
    subsort EmcyErrBody < EmcyErrNameSet .
    op nil : -> EmcyErrNameSet [ctor] .
    op __ : EmcyErrNameSet EmcyErrNameSet ->
    EmcyErrNameSet [ctor comm assoc id: nil] .


    msg Emergency_withDesc_ : EmcyErrCode EmcyErrBody -> Msg .
```

```
    class ErrType | validErrName : EmcyErrNameSet .
endom)


(omod NODE is
    including NAT .
    including EMERGENCY-OBJ .

    sort NodeStatus .
    op errorFree : -> NodeStatus .
    op errorOccured : -> NodeStatus .


    class Node |    ID : Nat,
                    errNo : Nat,
                    nodeState : NodeStatus .

    vars NodeO NodeO2 : Oid .
    vars O O2 : Oid .
    vars IDN IDN2 : Nat .
    vars N N2 : Nat .
    var NdStatus : NodeStatus .
    vars S S2 : EmcyErrNameSet .
    var EED : EmcyErrBody .

    rl[nodeErrorOccure] :
    < O : ErrType | validErrName : EED S >
    < NodeO : Node |    errNo : N,
                        nodeState : NdStatus >
    =>
    < NodeO : Node |    errNo : s(N),
                        nodeState : errorOccured >
    < O : ErrType | validErrName : EED S >
    (Emergency eecerr withDesc EED) .

    crl[oneErrorSovled] :
    < NodeO : Node |    errNo : s(N) >
    =>
```

```
    < NodeO : Node |    errNo : N >
    if N > 0 .

    rl[LastErrorSolved] :
    < NodeO : Node |    errNo : 1 >
    =>
    < NodeO : Node |    errNo : 0,
                        nodeState : errorFree >
   (Emergency eecclear withDesc clearErr) .
endom)


(omod NMT is
    protecting NAT .

    sort NetworkStatus .
    op initialising : -> NetworkStatus .
    op resetApp : -> NetworkStatus .
    op resetComm : -> NetworkStatus .
    op preOp : -> NetworkStatus .
    op operation : -> NetworkStatus .
    op stopped : -> NetworkStatus .

    sort NmtMsg .
    op node_bootUp : Oid -> NmtMsg .
    op changeNode_toState_ : Oid NetworkStatus -> NmtMsg .
    op changeAllNodesToState_ : NetworkStatus -> NmtMsg .

    sort NmtRole .
    op slave : -> NmtRole .
    op master : -> NmtRole .

    sort NodeStateRecord .
    op node_in_ : Oid NetworkStatus -> NodeStateRecord .

    sort NodeStateRecordList .
    subsort NodeStateRecord < NodeStateRecordList .
    op nilRdLt : -> NodeStateRecordList [ctor] .
```

```
    op __ : NodeStateRecordList NodeStateRecordList ->
    NodeStateRecordList [ctor comm assoc id: nilRdLt] .

    var NdRecd : NodeStateRecord .
    vars NtSt NtSt2 : NetworkStatus .
        vars OID OID2 : Oid .
    var NdStRecdLst : NodeStateRecordList .

    op updateRecord_toList_ :
     NodeStateRecord NodeStateRecordList ->
     NodeStateRecordList .
    eq updateRecord NdRecd toList nilRdLt = NdRecd .
    eq updateRecord (node OID in NtSt) toList (
     (node OID2 in NtSt2) NdStRecdLst) = if OID == OID2
     then
     (node OID2 in NtSt) NdStRecdLst
else
(node OID2 in NtSt2)
(updateRecord (node OID in NtSt)
toList NdStRecdLst)
fi .

    **** Update all nodes Status
    op update_toState_ :
     NodeStateRecordList NetworkStatus ->
     NodeStateRecordList .
    eq update nilRdLt toState NtSt = nilRdLt .
    eq update ((node OID in NtSt) NdStRecdLst)
     toState preOp =
     if NtSt == stopped or
     NtSt == operation
     then
     (node OID in preOp)
     (update NdStRecdLst toState preOp)
             else
              (node OID in NtSt) update
              NdStRecdLst toState preOp
    fi .
```

```
    eq update ((node OID in NtSt)
     NdStRecdLst) toState operation =
     if NtSt == stopped or NtSt == preOp
     then
     (node OID in operation) (update
     NdStRecdLst toState operation)
else
(node OID in NtSt) (update
NdStRecdLst toState operation)
fi .
    eq update ((node OID in NtSt)
    NdStRecdLst) toState stopped =
    if NtSt == operation or NtSt == preOp
    then
     (node OID in stopped)
     (update NdStRecdLst toState stopped)
    else
 (node OID in NtSt)
 (update NdStRecdLst toState stopped)
    fi .
    eq update ((node OID in NtSt) NdStRecdLst)
    toState resetApp =
    if
     NtSt == operation or NtSt == preOp or NtSt == stopped
    then
     (node OID in resetApp)
     (update NdStRecdLst toState resetApp)
    else
     (node OID in NtSt)
     (update NdStRecdLst toState resetApp)
    fi .
    eq update ((node OID in NtSt) NdStRecdLst)
    toState resetComm =
    if
     NtSt == operation or NtSt == preOp or NtSt == stopped
    then
     (node OID in resetComm) (update NdStRecdLst
     toState resetComm)
```

```
    else
      (node OID in NtSt)
      (update NdStRecdLst toState resetComm)
    fi .
endom)


(omod NODE is
    including NAT .
    including EMERGENCY-OBJ .
    including MESSAGE .
    including CONTROLLER .
    including MSG-RULES .
    including STRING .
    including NMT .

    sort NodeStatus .
    op errorFree : -> NodeStatus .
    op errorOccured : -> NodeStatus .

    subsort EmcyErrBody < FrameData MsgContent .
    subsort NmtMsg < FrameData .

    subsort String < Oid .

    class Node | ID : Nat,
                 controllerID : Oid,
                 errNo : Nat,
                 nodeState : NodeStatus,
                 networkState : NetworkStatus,
                 slaveRecdLst : NodeStateRecordList,
                 nmtRole : NmtRole .

    vars NodeO NodeO2 : Oid .
    vars CtrlO CtrlO2 : Oid .
    vars O O2 : Oid .
    vars FID FID2 : Oid .
    vars IDN IDN2 : Nat .
```

```
vars N N2 N3 N4 : Nat .
vars NID NID2 : Nat .
var NdStatus : NodeStatus .
vars S S2 : EmcyErrNameSet .
var EED : EmcyErrBody .
vars BUF BUF2 BUF3 BUF4 : Que .
var  FDATA : FrameData .
var NS : NatSet .
var F : Object .
var CONF : Configuration .
    vars StRecdLst StRecdLst2 : NodeStateRecordList .
vars NetworkState NetworkState2 : NetworkStatus .

op calCODByNodeID_andMsgTypeID_ : Nat Nat -> Nat .
eq calCODByNodeID NID andMsgTypeID N = NID * 10000 + N .


op pushMsg_toController_ : Object Object -> Object .
eq pushMsg F toController
 < Ctrl0 : Controller | txbuf : BUF,
 txbufCap : N2,
 failedTxbuf : BUF2,
 rxbuf : BUF3,
 rxbufCap : N3,
 failedRxbuf : BUF4,
 filter : NS >
 =
  if N2 > length(BUF) then
    < Ctrl0 : Controller |  txbuf : F addTo BUF,
                            txbufCap : N2,
                            failedTxbuf : BUF2,
                            rxbuf : BUF3,
                            rxbufCap : N3,
                            failedRxbuf : BUF4,
                            filter : NS >
  else
    < Ctrl0 : Controller |  txbuf : BUF,
                            txbufCap : N2,
```

```
                              failedTxbuf : BUF2 : F,
                              rxbuf : BUF3,
                              rxbufCap : N3,
                              failedRxbuf : BUF4,
                              filter : NS >
  fi .

rl [powerOn] :
< NodeO : Node | networkState : initialising >
=>
< NodeO : Node | networkState : resetApp > .

rl [initialization] :
< NodeO : Node | networkState : resetApp >
=>
< NodeO : Node | networkState : resetComm > .

rl [sendBootUp] :
< NodeO : Node | ID : NID,
                    controllerID : CtrlO,
                    errNo : 0,
                    nodeState : errorFree,
                    networkState : resetComm,
                    nmtRole : slave >
GLOBAL (CONF
        < CtrlO : Controller | >)
=>
< NodeO : Node | networkState : preOp >
GLOBAL (CONF
        (pushMsg < NodeO : Frame |
        id : calCODByNodeID NID andMsgTypeID 1792,
        other : (node NodeO bootUp) >
        toController < CtrlO : Controller | >)) .

rl [receiveBootUp] :
< NodeO : Node | ID : NID,
                    controllerID : CtrlO,
                    errNo : N,
```

```
                        nodeState : NdStatus,
                        slaveRecdLst : StRecdLst,
                        nmtRole : master >
    GLOBAL (CONF
            < Ctrl0 : Controller | rxbuf :
             (< FID : Frame | id : NID2,
             other : (node Node02 bootUp) > : BUF) >)
    =>
    < Node0 : Node | slaveRecdLst :
    (updateRecord (node Node02 in preOp) toList StRecdLst) >
    GLOBAL (CONF
            < Ctrl0 : Controller | rxbuf : BUF >) .

    crl [sendStartMsgToOneNode] :
        < Node0 : Node | nmtRole : master,
                         controllerID : Ctrl0,
                         ID : NID,
                         slaveRecdLst :
(StRecdLst2 (node Node02 in NetworkState) StRecdLst) >
        GLOBAL (CONF
                < Ctrl0 : Controller | >)
        =>
        < Node0 : Node |    slaveRecdLst :
        (StRecdLst2 (node Node02 in operation) StRecdLst) >
        GLOBAL (CONF
                (pushMsg < Node0 : Frame |
                id : (calCODByNodeID NID andMsgTypeID 0),
                other : (changeNode Node02 toState operation) >
                toController  < Ctrl0 : Controller | > ))
        if NetworkState == preOp or NetworkState == stopped .

    crl [receiveStartMsgFromMaster] :
        < Node0 : Node | nmtRole : slave,
                         controllerID : Ctrl0,
                         networkState : NetworkState >
        GLOBAL (CONF
                < Ctrl0 : Controller |
                rxbuf : < FID : Frame | id : NID,
```

```
            other : (changeNode Node0 toState operation) > :
BUF >)
     =>
     < Node0 : Node |    networkState : operation >
     GLOBAL (CONF
             < Ctrl0 : Controller | rxbuf : BUF >)
     if NetworkState == preOp or NetworkState == stopped .


 crl [receiveAllStartMsg] :
     < Node0 : Node | nmtRole : slave,
                      controllerID : Ctrl0,
                      networkState : NetworkState >
     GLOBAL (CONF
             < Ctrl0 : Controller | rxbuf :
             < FID : Frame | id : NID,
             other : (changeAllNodesToState operation) > :
             BUF >)
     =>
     < Node0 : Node |    networkState : operation >
     GLOBAL (CONF
             < Ctrl0 : Controller | rxbuf : BUF >)
     if NetworkState == preOp or NetworkState == stopped .

 crl [receiveNMTServiceWithUnmatchedID] :
 < Node0 : Node | nmtRole : slave,
                    controllerID : Ctrl0,
                    networkState : NetworkState >
 GLOBAL ( CONF
 < Ctrl0 : Controller | rxbuf : < FID : Frame | id : NID,
 other : (changeNode Node02 toState NetworkState2) > :
 BUF >)
 =>
 < Node0 : Node |    networkState : operation >
 GLOBAL (CONF
 < Ctrl0 : Controller | rxbuf : BUF >)
 if Node02 =/= Node0 .
```

```
    crl [sendStopRemoteNodeToSingleNode] :
        < NodeO : Node | nmtRole : master,
                         controllerID : CtrlO,
                         ID : NID,
                         slaveRecdLst : (StRecdLst2
(node NodeO2 in NetworkState) StRecdLst) >
        GLOBAL (CONF
        < CtrlO : Controller | >)
        =>
        < NodeO : Node | slaveRecdLst :
        (StRecdLst2 (node NodeO2 in stopped) StRecdLst) >
        GLOBAL (CONF
        (pushMsg < NodeO : Frame |
         id : (calCODByNodeID NID andMsgTypeID 0),
         other : (changeNode NodeO2 toState stopped) >
         toController  < CtrlO : Controller | > ))
        if NetworkState == preOp or NetworkState == operation .

    crl [sendStopRemoteNodeToAllNodes] :
        < NodeO : Node | nmtRole : master,
                         controllerID : CtrlO,
                         ID : NID,
                         slaveRecdLst : StRecdLst >
        GLOBAL (CONF
        < CtrlO : Controller | rxbuf : BUF >)
        =>
        < NodeO : Node | slaveRecdLst : update
        StRecdLst toState stopped >
        GLOBAL (CONF
        (pushMsg < NodeO : Frame |
        id : calCODByNodeID NID andMsgTypeID 0,
        other : (changeAllNodesToState stopped) >
        toController < CtrlO : Controller | > ))
        if StRecdLst =/= nilRdLt .

    crl [receiveStopRemoteNodeMsg] :
        < NodeO : Node | nmtRole : slave,
                         controllerID : CtrlO,
```

```
                        networkState : NetworkState >
        GLOBAL (CONF
        < Ctrl0 : Controller | rxbuf :
        < FID : Frame | id : NID,
        other : (changeNode Node0 toState stopped) > :
        BUF >)
        =>
        < Node0 : Node | networkState : stopped >
        GLOBAL (CONF
        < Ctrl0 : Controller | rxbuf : BUF >)
        if NetworkState == preOp or NetworkState == operation .

    crl [receiveStopAllNodesMsg] :
        < Node0 : Node |  nmtRole : slave,
                        controllerID : Ctrl0,
                        networkState : NetworkState >
        GLOBAL (CONF
        < Ctrl0 : Controller | rxbuf :
        < FID : Frame | id : NID,
        other : (changeAllNodesToState stopped) > : BUF >)
        =>
        < Node0 : Node | networkState : stopped >
        GLOBAL (CONF
        < Ctrl0 : Controller | rxbuf : BUF >)
        if NetworkState == preOp or NetworkState == operation .

    crl [sendEnterPreMsgToSingleNode] :
        < Node0 : Node | nmtRole : master,
                        controllerID : Ctrl0,
                        ID : NID,
                        slaveRecdLst :
(StRecdLst2 (node Node02 in NetworkState) StRecdLst) >
        GLOBAL (CONF
        < Ctrl0 : Controller | >)
        =>
        < Node0 : Node | slaveRecdLst :
        (StRecdLst2 (node Node02 in preOp) StRecdLst) >
        GLOBAL (CONF
```

```
    (pushMsg < NodeO : Frame |
    id : (calCODByNodeID NID andMsgTypeID 0),
    other : (changeNode NodeO2 toState preOp) >
    toController  < CtrlO : Controller | > ))
    if NetworkState == stopped or NetworkState == operation .

crl [sendEnterPreMsgToAllNodes] :
    < NodeO : Node | nmtRole : master,
                     controllerID : CtrlO,
                     ID : NID,
                     slaveRecdLst : StRecdLst >
    GLOBAL (CONF
    < CtrlO : Controller | rxbuf : BUF >)
    =>
    < NodeO : Node | slaveRecdLst :
     update StRecdLst toState preOp >
    GLOBAL (CONF
    (pushMsg < NodeO : Frame |
    id : calCODByNodeID NID andMsgTypeID 0,
    other : (changeAllNodesToState preOp) >
    toController < CtrlO : Controller | > ))
    if StRecdLst =/= nilRdLt .

crl [receiveEnterPreMsg] :
    < NodeO : Node | nmtRole : slave,
                     controllerID : CtrlO,
                     networkState : NetworkState >
    GLOBAL (CONF
    < CtrlO : Controller | rxbuf :
    < FID : Frame | id : NID,
    other : (changeNode NodeO toState preOp) > :
    BUF >)
    =>
    < NodeO : Node | networkState : preOp >
    GLOBAL (CONF
    < CtrlO : Controller | rxbuf : BUF >)
    if NetworkState == stopped or NetworkState == operation .
```

```
    crl [receiveAllEnterPreMsg] :
        < NodeO : Node | nmtRole : slave,
                         controllerID : CtrlO,
                         networkState : NetworkState >
        GLOBAL (CONF
        < CtrlO : Controller | rxbuf :
        < FID : Frame | id : NID,
        other : (changeAllNodesToState preOp) > : BUF >)
        =>
        < NodeO : Node | networkState : preOp >
        GLOBAL (CONF
        < CtrlO : Controller | rxbuf : BUF >)
        if NetworkState == stopped or NetworkState == operation .

    crl [sendResetAppMsgToSingleNode] :
        < NodeO : Node | nmtRole : master,
                         controllerID : CtrlO,
                         ID : NID,
                         slaveRecdLst :
(StRecdLst2 (node NodeO2 in NetworkState) StRecdLst) >
        GLOBAL (CONF
        < CtrlO : Controller | >)
        =>
        < NodeO : Node | slaveRecdLst :
        (StRecdLst2 (node NodeO2 in resetApp) StRecdLst) >
        GLOBAL (CONF
        (pushMsg < NodeO : Frame |
        id : (calCODByNodeID NID andMsgTypeID 0),
        other : (changeNode NodeO2 toState resetApp) >
        toController  < CtrlO : Controller | > ))
        if NetworkState == stopped or
        NetworkState == operation or NetworkState == preOp .

    crl [sendResetAppMsgToAllNodes] :
        < NodeO : Node | nmtRole : master,
                         controllerID : CtrlO,
                         ID : NID,
                         slaveRecdLst : StRecdLst >
```

```
    GLOBAL (CONF
    < Ctrl0 : Controller | rxbuf : BUF >)
    =>
    < Node0 : Node | slaveRecdLst :
     update StRecdLst toState resetApp >
    GLOBAL (CONF
    (pushMsg < Node0 : Frame |
    id : calCODByNodeID NID andMsgTypeID 0,
    other : (changeAllNodesToState resetApp) >
    toController < Ctrl0 : Controller | > ))
    if StRecdLst =/= nilRdLt .

crl [receiveResetAppMsg] :
    < Node0 : Node |  nmtRole : slave,
                      controllerID : Ctrl0,
                      networkState : NetworkState >
    GLOBAL (CONF
    < Ctrl0 : Controller | rxbuf : < FID : Frame | id : NID,
    other : (changeNode Node0 toState resetApp) > :
    BUF >)
    =>
    < Node0 : Node | networkState : resetApp >
    GLOBAL (CONF
    < Ctrl0 : Controller | rxbuf : BUF >)
    if NetworkState == stopped or
    NetworkState == operation or NetworkState == preOp .


crl [receiveResetAllAppMsg] :
    < Node0 : Node | nmtRole : slave,
                      controllerID : Ctrl0,
                      networkState : NetworkState >
    GLOBAL (CONF
    < Ctrl0 : Controller | rxbuf : < FID : Frame | id : NID,
    other : (changeAllNodesToState resetApp) > :
    BUF >)
    =>
    < Node0 : Node | networkState : resetApp >
```

```
        GLOBAL (CONF
        < Ctrl0 : Controller | rxbuf : BUF >)
        if NetworkState == stopped or
        NetworkState == operation or NetworkState == preOp .

    crl [sendResetAppMsgToSingleNode] :
        < Node0 : Node | nmtRole : master,
                         controllerID : Ctrl0,
                         ID : NID,
                         slaveRecdLst :
(StRecdLst2 (node Node02 in NetworkState) StRecdLst) >
        GLOBAL (CONF
        < Ctrl0 : Controller | >)
        =>
        < Node0 : Node | slaveRecdLst :
(StRecdLst2 (node Node02 in resetApp) StRecdLst) >
        GLOBAL (CONF
        (pushMsg < Node0 : Frame |
        id : (calCODByNodeID NID andMsgTypeID 0),
        other : (changeNode Node02 toState resetApp) >
        toController  < Ctrl0 : Controller | > ))
        if NetworkState == stopped or
        NetworkState == operation or NetworkState == preOp .

    crl [sendResetAppMsgToAllNodes] :
        < Node0 : Node | nmtRole : master,
                         controllerID : Ctrl0,
                         ID : NID,
                         slaveRecdLst : StRecdLst >
        GLOBAL (CONF
        < Ctrl0 : Controller | rxbuf : BUF >)
        =>
        < Node0 : Node | slaveRecdLst : update
         StRecdLst toState resetApp >
        GLOBAL (CONF
        (pushMsg < Node0 : Frame |
        id : calCODByNodeID NID andMsgTypeID 0,
        other : (changeAllNodesToState resetApp) >
```

```
         toController < Ctrl0 : Controller | > ))
        if StRecdLst =/= nilRdLt .

    crl [receiveResetCommMsg] :
        < Node0 : Node | nmtRole : slave,
                         controllerID : Ctrl0,
                         networkState : NetworkState >
        GLOBAL (CONF
        < Ctrl0 : Controller | rxbuf : < FID : Frame |
        id : NID,
        other : (changeNode Node0 toState resetComm) > :
        BUF >)
        =>
        < Node0 : Node | networkState : resetComm >
        GLOBAL (CONF
        < Ctrl0 : Controller | rxbuf : BUF >)
        if NetworkState == stopped or
        NetworkState == operation or NetworkState == preOp .

    crl [receiveResetAllComm] :
        < Node0 : Node | nmtRole : slave,
                         controllerID : Ctrl0,
                         networkState : NetworkState >
        GLOBAL (CONF
        < Ctrl0 : Controller | rxbuf : < FID : Frame | id : NID,
        other : (changeAllNodesToState resetComm) > :
        BUF >)
        =>
        < Node0 : Node | networkState : resetComm >
        GLOBAL (CONF
        < Ctrl0 : Controller | rxbuf : BUF >)
        if NetworkState == stopped or
        NetworkState == operation or NetworkState == preOp .
endom)
```

Bibliography

[1] Brandt W, Dang AS, Magne E, Crowley D, Houston K, Rennie A, Hodder M, Stringer R, Juiniti R, Ohara S, Rushton S. *Deepening the Search for Offshore Hydrocarbons*. Oilfield Review 10, no. 1, 1998.

[2] Alan Christie, Ashley Kishino, John Cromb, Rodney Hensley, Ewan Kent, Brian McBeath, Hamish Stewart, Alain Vidal, Leo Koot. *Subsea Solutions*. http://www.slb.com/~/media/Files/resources/oilfield_review/ors99/win99/composite.pdf, April 19, 2014

[3] M. Clavel, F. Duràn, S. Eker, P. Lincoln, N. Martì-Oliet, J. Meseguer, J.F. Quesada. *Maude: specification and programming in rewriting logic, Manual distributed as documentation of the Maude system*. Computer Science Laboratory, SRI International, http://maude.cs.uiuc.edu/overview.html, April 14, 2014.

[4] Peter C. Ölveczky. *Modeling and Analyzing Protocols in Maude and Real-Time Maude Tutorial*. http://www2.ic.uff.br/~cbraga/vas/rtmaudetutorial.html, April 14, 2014.

[5] David Saul. *Subsea instrumentation interface standardization in the offshore oil and gas industry*. BP Exploration Operating Co Ltd - EPTG Offshore Systems Team, Sunbury UK, http://www.can-cia.org/fileadmin/cia/files/icc/11/saul.pdf, 2006.

[6] CAN in Automation (CiA). *CANOpen application layer and communication profile*, CiA 301 V4.2.0. 21 Feb 2011.

[7] K. Etschberger. *Controller Area Network: Basics, Protocols, Chips and Applications*. IXXAT Automation GmbH, Weingarten, Germany. 2001.

[8] Olaf Pfeiffer, Andrew Ayre, Christian Keydel, *Embedded Networking with CAN and CANopen*. Copperhill Media, Jan 1, 2008.

[9] CAN in Automation (CiA). *CAN history*. http://www.can-cia.org/index.php?id=522#c2106, April 16, 2014

[10] CAN in Automation (CiA). *The CANOpen Object Dictionary*. http://www.can-cia.org/index.php?id=506, April 19, 2014.

[11] H.Boterenbrood. *CANopen, high-level protocol for CAN-bus*. NIKHEF internal documentation, NIKHEF, Amsterdam Version 3.0, March 20, 2000.

[12] CAN in Automation (CiA). *SYNC Protocol*. http://www.can-cia.org/index.php?id=206, April 19, 2014

[13] CAN in Automation (CiA), *Time-stamp Protocol*. http://www.can-cia.org/index.php?id=210, April 19, 2014

[14] Peter Csaba Ölveczky, *Formal Modeling and Analysis of Distributed Systems in Maude*. http://folk.uio.no/peterol/inf3230-lecturenotes.html, April 19, 2014

[15] Computer Science Library SRI International, Department of Computer Science in University of Illinois at Urbana-Champaign. *The Maude System*. http://maude.cs.uiuc.edu, April 19, 2014

[16] F. Duràn and J. Meseguer. *An extensible module algebra for Maude*. Second International Workshop on Rewriting Logic and its Applications, volume 15 of Electronic Notes in Theoretical Computer Science, 1998.

[17] Santiago Escobar, Josè Meseguer, Prasanna Thati. *Narrowing and Rewriting Logic: from Foundations to Applications*. Proceedings of the 15th Workshop on Functional and (Constraint) Logic Programming, volume 177 of Electronic Notes in Theoretical Computer Science, 2007

[18] Steven Eker, Josè Meseguer, Ambarish Sridharanarayanan. *The Maude LTL Model Checker*. 4th International Workshop on Rewriting Logic and its Applications, volume 71 of Electronic Notes in Theoretical Computer Science, 2001.

[19] M. Farsi, K. Ratcliff, Manuel Barbosa. *An Overview of Control Area Network*. Computing & Control Engineering Journal, volume10, issue: 3, 1999.

[20] M. Barbosa, M. Farsi, C. Allen, A.S. Carvalho. *Formal Validation of the CAN-Open Communication Protocol*. Fieldbus Systems and Their Applications 2003, 2003.

[21] Manuel Clavel, Francisco Duràn, Joe Hendrix, Salvador Lucas, Josè Meseguer, Peter Ölveczky. *The Maude Formal Tool Environment*. Algebra and Coalgebra in Computer Science, 2007.

[22] Mirko Tischer, Dietmar Widmann, Vector Informatik GmbH. *Model based testing and Hardware-in-the-Loop simulation of embedded CANopen control devices*. International CAN Conference 2012, 2012.

[23] G. J. Holzmann. *The model checker SPIN*. IEEE Trans. on Software Engineering, 1997.